Data Structures and Algorithms

Dr. Yan Huang

Course Description

Title: Data Structures and Algorithms
Lecturer: Dr. Coral Yan Huang
Email: Yan.Huang@cs.cf.ac.uk
Room: T2.07
Time: 11:10-13:00 Thursday
Handouts: Lecture notes,
Exercise sheets (available only at lectures)

Coursework and Exams

Coursework	Given out: week 6 Due in: Week 10	12% weight
Mid-term Class Test	Week 6	8% weight
Final Exam		80% weight

Books

The recommended text book for the module is:

"***Data structures and algorithms in Java***"
3rd edition, Goodrich and Tamassia
Wiley, 2003

Check reading list for course at
http://www.readinglists.co.uk/

Student view password is: *CMT502YH*

And more books…

- More recommendations are:
  - **"Data Structures and Algorithms in Java"** by Robert Lafore, published by SAMS Second Education, 2003
  - **"Data Structures and Other Objects Using Java"** by Michael Main, published by Addison Wesley Publishing Company, 2002

Scheduling (to be continued)

Week	Topic
1	Chapter 1. Introduction Chapter 2. Analysis Tools
2	Chapter 3. Stacks Chapter 4. Queues
3	Chapter 5. LinkedLists Chapter 6. Lists
4	Chapter 7. Trees
5	Chapter 7. Trees
6	Half-term Test Coursework given out

Scheduling

Week	Topic
7	Chapter 8. Balanced search trees Chapter 9. Sorting
8	Chapter 9. Sorting
9	Chapter 10. Searching and selection Chapter 11. Maps, dictionaries and sets
10	Chapter 12. Graphs Coursework due
11	Chapter 13. Greedy Algorithms
12	Revision

Chapter 1: Introduction

- 1.1 Algorithms
  - 1.1.1 Definition and Properties
  - 1.1.2 An Example
  - 1.1.3 Analysis of Algorithms
  - 1.1.4 Pseudocode for Algorithms
- 1.2 Data Structures
  - 1.2.1 Definition
  - 1.2.2 Relation to Algorithms
  - 1.2.3 Data Structure Types

1.1 Algorithms

Computer Science = Study of Algorithms

- Machines for executing algorithms
- +
- Language for describing algorithms
- +
- Foundations of algorithms
- +
- Analysis of algorithms

1.1.1 Definition and Properties

- **Definition**: An algorithm is a finite set of instructions which accomplish a particular task.
- **Properties:**
  - Input
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness

1.1.2 An Example

- **Example 1.1**. Finding the maximum of three numbers a, b and c.

```
Input: a, b, c
Output: x

Max(a, b, c){
        x=a;
        if ( b > x )
                x = b
        if (c > x )
                x = c
        return x
}
```


1.1.3 Analysis of algorithms

- Correctness
- Termination
- Time analysis: How many instructions does the algorithm execute?
- Space analysis: How much memory does the algorithm need to execute?

1.1.4 Pseudo code for Algorithms

- An algorithm contains
  - Declaration of input and output.
  - Functions
- Pseudocode syntax
  - If statements

```
if (condition)
    action
```

```
if (condition)
    action1
then
    action2
```


1.1.4 Pseudo code for Algorithms

- While statement

```
While (condition)
    action
```

- For statement

```
for (var=init; var<=limit; var++)
    action
```

- Return statement

```
return x
```

- Operators

Arithmetic operators:	+, -, *, /
Relational operators:	==, !=, >, <, >=, <=
Logical operators:	&&, ||, !
Assignment operator:	=

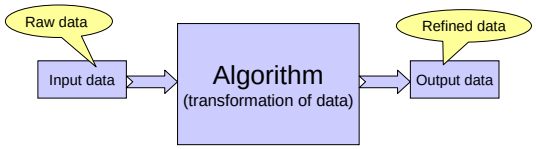
1.1.4 Pseudocode for Algorithms

- **Example 1.2**: Finding the maximum value in an Array.

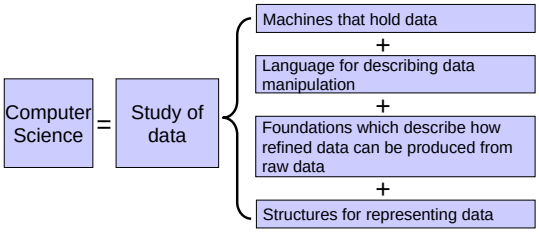
```
Input: s
Output: x

ArrayMax(s){
        x=s[0]
        for(i=1; i<s.length; i++)
          if (s[i] > x )
                x = s[i]
        return x
}
```


1.2 Data Structures

- We might also say computer science is the study of data



1.2 Data Structure



1.2.1 Definition

- A data structure is an organization of information, usually in memory, for better algorithm efficiency.
- A data structure may include redundant information, such as length of the list or number of nodes in a tree.

1.2.2 Relation to Algorithms

- Most data structures have associated algorithms to perform operations, such as search, insert, or balance, that maintain the properties of the data structure
- Algorithms and data structures should be thought of as a unit, neither one making sense without the other.

1.2.3 Data Structure Types

- Queue
- Stack
- Linked List
- Heap
- Dictionary
- Tree
- Conceptual unity
- ……

Chapter 2. Analysis Tools

- 2.1 Mathematical review
  - 2.1.1 Exponents
  - 2.1.2 Logarithms
  - 2.1.3 Summations
- 2.2 Running time
- 2.3 Analysis of algorithms
  - 2.3.1 Primitive operations
  - 2.3.2 Average case and worst case analysis
- 2.4 Big-O notation

2.1 A Quick Mathematical Review

- 2.1.1 Exponents
  - The number being multiplied is called the base, and the exponent tells how many times the base is multiplied by itself.
    $4 \times 4 \times 4 \times 4 \times 4 \times 4 = 4^6$
  - Propositions
    - $(b^a)^c = b^{ac}$
    - $b^a b^c = b^{a+c}$
    - $b^a / b^c = b^{a-c}$

2.1 A Quick Mathematical Review

- 2.1.2 Logarithms
  - A logarithmic function is the inverse of an exponential function
    $\log_b a = c$ if $a = b^c$
  - Propositions
    - $\log_b(ac) = \log_b a + \log_b c$
    - $\log_b(a/c) = \log_b a - \log_b c$
    - $\log_b a^c = c \log_b a$

2.1 A Quick Mathematical Review

- 2.1.3 Summations

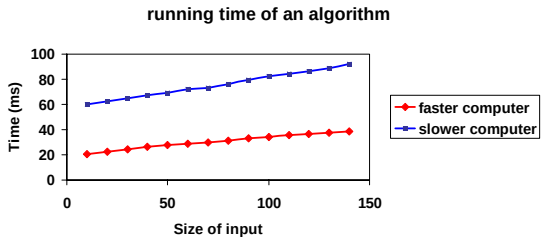
$$(1)\ \sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

$$(2)\ \sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

2.2 Running Time

- Running time — the actual time spent in execution of an algorithm.
- It depends on a number of factors
  - Input
  - The hardware and software environment.

2.2 Running Time



2.2 Running Time

- General methodology required
  - Take into account all possible inputs
  - Independent from the hardware and software environment.
- We conclude
  - Experimental analysis has its limitations
  - It is better to analyse a particular algorithm without performing experiments on its running time.

2.3 Analysis of Algorithms

- Instead of trying to determine the specific execution time of a particular algorithm, we simply count the number of primitive operations that are executed

2.3.1 Primitive Operations

- Includes
  - Assigning a value to a variable
  - Calling a method
  - Performing an arithmetic operation
  - Comparing two values
  - Indexing into an array
  - Following an object reference
  - Returning from a method

2.3.1 Primitive Operations

- **Example 2.1**: Given an algorithm which finds the maximum value in an array, count the number of primitive operations executed in this algorithm.

```
Input: s
Output: x

ArrayMax(s){
        x=s[0]
        for(i=1; i<s.length; i++)
          if (s[i] > x )
                x = s[i]
        return x
}
```

2.3.1 Primitive Operations

```
ArrayMax(s){
  x=s[0]
  for(i=1; i<s.length; i++)
     if (s[i] > x )
          x = s[i]
  return x
}
```

2 operations (indexing and assignment)

Loop
- Beginning of loop: 1 assignment
- On entering each iteration (n): 1 comparison
- Each iteration (n-1):
  - 2 operations(indexing and comparison)
  - 0/2 operations(indexing and assignment)
- End of each iteration (n-1): 2 operations (summing and assignment)

1 operation (returning)

(n is the size of the input array)

2.3.1 Primitive Operations

- The number of primitive operations *t(n)* executed by algorithm *arrayMax* is (n is the size of the input array):

<u>At least</u>

t(n) = 2+1+n+(2+2)(n-1)+1 = 5n* — Best case

<u>At most</u>

t(n) = 2+1+n+(2+2+2)(n-1)+1 = 7n-2* — Worst case

2.3.2 Average-Case and Worst-Case Analysis

- Average-Case Analysis—expresses the running time of an algorithm as an average taken over all possible inputs.
  - Difficult — depends on the input distribution, and requires heavy mathematics on probability theory.
  - Not required.

2.3.2 Average-Case and Worst-Case Analysis

- Best-Case Analysis — the shortest running time of an algorithm.
- Worst-Case Analysis — the longest running time of an algorithm.

2.4 “Big-O” Notation

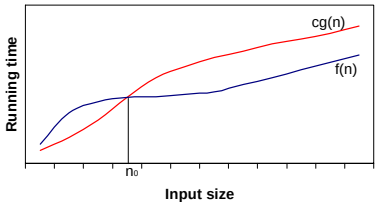
- Definition of “Big-O” Notation(a very mathematical one)

  Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that f(n) is O(g(n)) if there is a real constant c>0 and an integer constant n0 >= 1 such that f(n)<=cg(n) for every integer n>= n0.

“f(n) is O(g(n))” is pronounced as
“f(n) is big-O of g(n)” or “f(n) is order g(n)”.

2.4 “Big-O” Notation

f(n) is O(g(n))



2.4 "Big-O" Notation

Example 2.2: Justify 7n-2 is O(n).

Justification: we need to find a real constant c>0 and an integer n0>=1 such that 7n-2<=cn for every n>=n_0.

We chose c=7, n0=1 and then we have

7n-2<7n when n>=1

Thus 7n-2 is O(n)

2.4 "Big-O" Notation

Example 2.3: Justify $20n^3+5n^2+6$ is $O(n^3)$.

Justification:

We chose c=31, n0=1 and then we have

$20n^3+5n^2+6 \le 31n^3$ when n>=1

Thus $20n^3+5n^2+6$ is $O(n^3)$.

2.4 "Big-O" Notation

- A practical method in finding the "big-O" notation of a function.
  - We can use the notion of the largest term in a function.
  - The largest term is
    - the term with the largest exponent of n
    - the term that grows the fastest.

2.4 "Big-O" Notation

- **Example 2.4:** 5n+logn+7 is O(n)

n	Log n
1	0
10	1
100	2
1000	3
…	…

- **Example 2.5:** 100 is O(1)
- **Example 2.6:** 5/n is O(1/n)

2.4 "Big-O" Notation

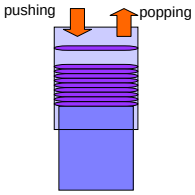
- **Special terms to classify functions**
  - Logarithmic functions: O(logn)
  - Linear functions: O(n)
  - Quadratic functions: $O(n^2)$
  - Polynomial functions: $O(n^k)$ where k>=1
  - Exponential functions: $O(a^n)$ where a>1

Chapter 3 Stacks

- 3.1 The stack Abstract Data Type
- 3.2 Array-Based Implementation of Stack
- 3.3 Stack Applications
- 3.4 Stacks in the Java Virtual Machine
  - 3.4.1 Java Method Stack
  - 3.4.2 Recursion
  - 3.4.3 Operand Stacks

Chapter 3. Stacks

- Definition — A stack is a container of objects that are inserted and removed according to the last-in-first-out principle.
- Operations:
  - Pushing
  - Popping
- Examples
  - "back" in a web browser
  - "undo" in Text editors



3.1 The stack Abstract Data Type

- Two fundamental operations
  - push (o): insert object o at the top of the stack
  - pop(o): return and remove the top object from the stack.

3.2 Array-Based Implementation of Stack

```
public class ArrayStack{

        public static int capacity;
        public Object s[]; //Array used to implement the stack
        private int top=-1; //Index of the top element

        public ArrayStack(int n){
                capacity = n;
                s = new Object[n];
        }

        public void push(Object obj) { … }

        public Object pop() { … }
}
```


3.2 Array-Based Implementation of Stack

```
public class ArrayStack{
        ….
        public void push(Object obj) {
                if (top==capacity-1){
                        //stack is full , do nothing   2
                } else {
                        top++;                          2
                        s[top]=obj;                     2
                }
                return;                                 1
        }
        ….
}
```


3.2 Array-Based Implementation of Stack

```
public class ArrayStack{
        ….
        public Object pop() {
                Object ret=null;
                if (top<0){                             1
                        //stack is empty, do nothing
                } else {
                        ret = s[top];                   2
                        s[top]=null;                    2
                        top--;                          2
                }                                       1
                return ret;
        }
        ….
}
```


3.2 Array-Based Implementation of Stack

- Running time analysis

method	Number of primitive operations	Time
Push	7	O(1)
pop	8	O(1)

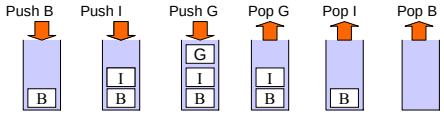
  - Both methods run in constant time O(1)
- Space usage: O(n)

3.2 Array-Based Implementation of Stack

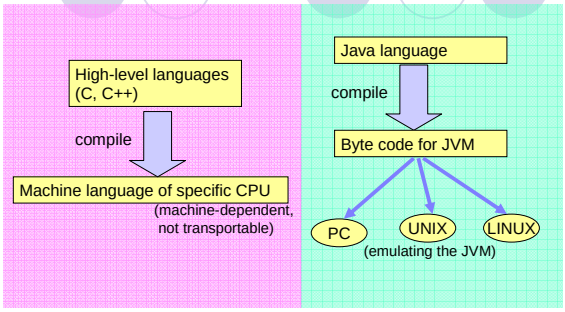
- Pros
  - Simple and Efficient
- Cons
  - May waste memory if the actually used space is smaller than the ultimate size of the stack.
  - May "crash" the applications which need a large size of stack.
  - Difficult to delete or insert an element.

3.3 Stack Applications

Example 3.1: Show how to use a stack to reverse a word BIG.



3.4 Stack in Java Virtual Machine



Typical Case Java Virtual Machine

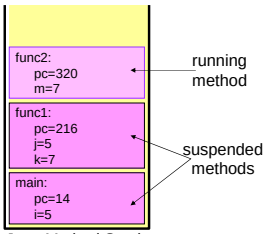
3.4.1 Java Method Stack

- Is a private stack for each running Java Program.
- Is used to keep track of important information on methods which includes
  - Local variables
  - Program counter
  - Parameters
- Its element is called frame which is a descriptor of one of active invocations of methods.

3.4.1 Java Method Stack

```
0     main(){
1         int i=5;
          …
14        func1(i)
          …
      }

204   func1(int j){
          int k=7;
          …
216       func2(k);
          …
      }

320   func2( int m){
          …
      }
```

Java Program



Java Method Stack

3.4.2 Recursion

- Allows a method to call itself as a subroutine.

Example: Compute the factorial function

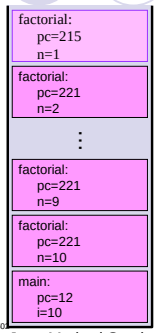
*n!=n(n-1)(n-2)…..*2*1*

```
Public static long factorial(long n){
   if(n<=1)
     return 1;
   else{
     int m=factorial(n-1);
     return n*m;
   }
}
```

3.4.2 Recursion

```
        main(){
            int i=10;
      .....
12    factorial(i);
      .....
        }

215   Public static long
      factorial(long n)
        {
      if(n<=1)
                return 1;
          else {
221     int m=factorial(n-1);
        return n*m;
          }
      }
```

Java Program



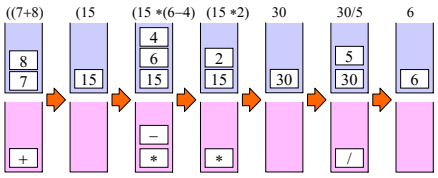
Java Method Stack

3.4.3 Operand Stacks

- are used to evaluate arithmetic expressions.
- Two kind of stacks work together for the purpose.
  - Number stack
  - Operation stack

3.4.3 Operand Stacks

Example 3.2 *((7+8)*(6-4))/5*

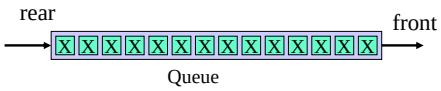


Chapter 4 Queues

- 4.1 The Queue Abstract Data Type
- 4.2 Array-Based Implementation of Queue
- 4.3 Priority Queues

Chapter 4 Queues

- Queue is a container of objects that are inserted and removed according to the first-in-first-out (FIFO) principle.



4.1 The Queue Abstract Data Type

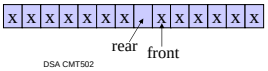
- Two fundamental operations
  - enqueue (o): insert object o at the rear of the queue
  - dequeue(o): return and remove from the queue the object at the front.

4.2 Array-Based Implementation of Queue

```
public class ArrayQueue{
        public static int capacity;
        public Object s[];          //Array used to implement the queue
        private int front = 0;      //Index of the front element
        private int rear = 0;       //index of the rear element

        public ArrayQueue(int n){
                capacity = n;
                s = new Object[n];
        }

        public void enqueue(Object obj) { … }

        public Object dequeue() { … }
}
```

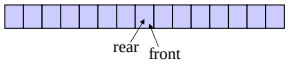

4.2 Array-Based Implementation of Queue

```
public class ArrayQueue{
        ….

        public void enqueue(Object obj) {
                if( (rear+1)%capacity == front )        3
                        // the queue is full, do nothing
                } else {
                        s[rear]=obj;                    2
                        rear=(rear+1) % capacity;       3
                }
                return;                                 1
        }

        …..
}
```



4.2 Array-Based Implementation of Queue

```
public class ArrayQueue{
        ….
        public Object dequeue() {
                Object retObj=null;                     1
                if( rear == front)                      1
                        // the queue is empty, do nothing
                } else {
                        retObj = s[front];              2
                        s[front]=null;                  2
                        front=(front+1)%capacity;       3
                }                                       1
                return retObj;
        }
        …..
}
```



4.2 Array-Based Implementation of Queue

- Running time analysis

method	Number of primitive operations	Time
enqueue	9	O(1)
dequeue	10	O(1)

  - Both methods run in constant time O(1)
- Space usage: O(n)

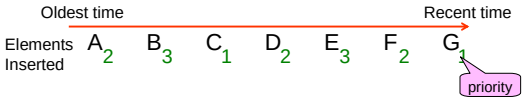
4.2 Array-Based Implementation of Queue

- Pros
  - Simple and Efficient
- Cons
  - Not very adaptive. The size must be fixed in advance.
  - Difficult to delete or insert an element.

4.3 Priority Queues

- A priority queue, in general, is a collection of prioritized elements, in which the next element to be removed in the queue is the element that
  - Has the highest priority of all elements
  - Has been in the queue the longest among elements with equal priority

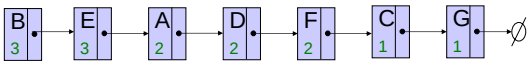
4.3 Priority Queues



- Removal order: B, E, A, D, F, C, G

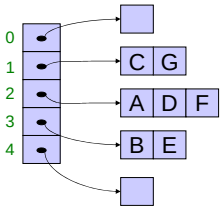
4.3 Priority Queues

- Representing priority queues
  - A linked list representation



4.3 Priority Queues

- A array of queues representation

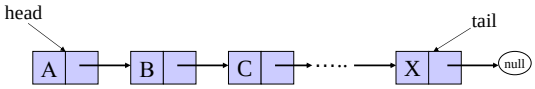


Chapter 5 Linked Lists

- 5.1 Singly Linked Lists
  - 5.1.1 Why Linked Lists
  - 5.1.2 Linked List ADT
  - 5.1.3 Implementing a singly Linked List
  - 5.1.4 Implementing a Stack with a Singly Linked List
- 5.2Doubly Linked List

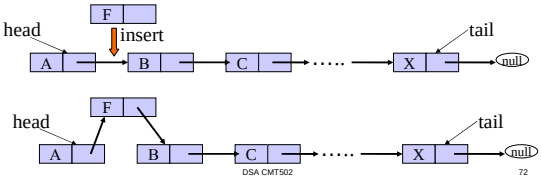
5.1 Singly Linked Lists

- Is a collection of nodes that together form a linear ordering. Each node is a compound object that stores an element and a reference, called next, to another node.



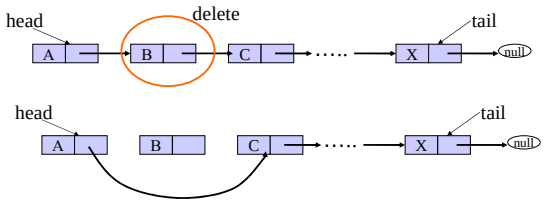
5.1.1 Why Linked Lists

- Why?
  - No predetermined fixed size.
  - Easily insert or delete an element.
- Insert an element



5.1.1 Why Linked Lists

- Delete an element



5.1.2 Linked List ADT

(Based on java.util.LinkedList)

- Fundamental methods
  - Add an element
    - add(int index, Object element)
    - addFirst(Object element)
    - addLast(Object element)
  - Remove an element
    - Object remove(int index)
    - boolean remove(Object element)
    - Object removeFirst()
    - Object removeLast()

5.1.2 Linked List ADT

(Based on java.util.LinkedList)

- Get an element
  - Object get(int index)
  - Object getFirst ()
  - Object getLast()
- Set an element
  - set(int index, Object element)
- Find the index of an element
  - int indexOf(Object element)
  - Int lastIndexOf (Object element)

5.1.2 Linked List ADT

(Based on java.util.LinkedList)

- Other methods
  - clear()
  - boolean contains(Object element)
  - int size()

5.1.3 Implementing a singly Linked List

- 5.1.3.1 Declaring a Node Class

```
public class Node{
    Public Object element = null;
    public Node next = null;

    public Node(Object o, Node n){
        element = o;
        next = n;
    }
}
```


5.1.3.2 Declaring the singly Linked List

```
public class SinglyLinkedList{
    private Node head=null;
    private Node tail=null;
    private int size=0;

  public SinglyLinkedList(){ }

    public void add(int index, Object element){..}
    public void addFirst(Object element){..}

    …

}
```

5.1.3.3 Declaring add methods

```
public void add(int index, Object element){
        Node newNode = new Node(element, null);
        if (index==0) {
                addFirst(element);   return;
        } else if (index >= size) {
                addLast(element);   return;
        }
        int k=0;
        Node node = this.head;
        while(k<index-1){
                node=node.next;
                k++;
        }
        newNode.next=node.next;
        node.next=newNode;
        size++;
        return;
}
```


5.1.3.3 Declaring add methods

```
public void addFirst(Object element){
        Node newNode = new Node(element, head);
        head = newNode;
        if(size==0) tail=newNode;
        size++;
}
public void addLast(Object element){
        Node newNode = new Node(element, null);
        if(tail!=null){
                tail.next=newNode;
                tail = tail.next;
        }else {
                tail=newNode;
                head=newNode;
        }
        size++;
}
```


5.1.3.4 Declaring remove methods

```
public Object remove (int index){
        int k=0;
        Node node1 = this.head;
        Node node2 = null;

        while(k<index && node1!=null){
                node2=node1;
                node1=node1.next;
                k++;
        }
        if(k==index) {
                node2.next = node1.next;
                size--;
                if(node2.next==null)
                    this.tail = node2;
                return node1.element;
        }
        return null;
}
```


5.1.3.4 Declaring remove methods

```
public Object removeFirst(){
    if(size==0) return null;
    Object retE = head.element;
    head=head.next;
    size--;
    if(size==0) tail = null;
    return retE;
}
```


5.1.3.4 Declaring remove methods

```
public Object removeLast(){
        if(size==0) return null;
        Object retE = tail.element;
        Node node1 = head;
        Node node2 = null;
        while(node1.next!=null){
                node2 = node1;
                node1=node1.next;
        }
        if(node2 == null)
                head=tail=null;
        else {
                node2.next=null;
                tail = node2;
        }
        size--;
        return retE;
}
```


5.1.3.4 Algorithm Analysis

Method	Time
add(int index, Object O)	O(n)
addFirst(Object O)	O(1)
addLast(Object O)	O(1)
remove(int index)	O(n)
removeFirst()	O(1)
removeLast()	O(n)

5.1.4 Implementing a Stack with a Singly Linked List—method 1

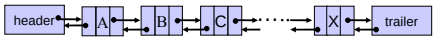
```
Public class LinkedStack{
    private Node top = null;
    public LinkedStack(){}

    public void push(Object element){
        Node node = new Node(element, top);
        top = node;
    }

    public Object pop(){
        if(top==null) return null;
        Object retObj = top.element;
        top = top.next;
        return retObj;
}
```


5.1.4 Implementing a Stack with a Singly Linked List—method 2

```
Public class LinkedStack{
    private SinglyLinkedList linkedList = null;

    public LinkedStack(){
        this.linkedList = new SinglyLinkedList();
    }

    public void push(Object element){
        this.linkedList.addFirst(element);
    }

    public Object pop(){
        return this.linkedList.removeFirst();
}
```


5.2 Doubly Linked List

- A node stores two references: *next* and *prev.*
- Two sentinel nodes
  - Header — null element, null prev.
  - Trailer — null element, null next.
- Why — insertion and removal at both ends run in O(1) time.



5.2 Doubly Linked List

- Implementation of a node

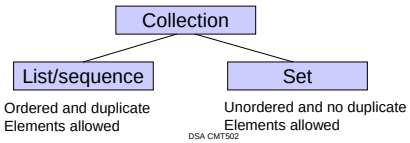
```
public class DLNode{
    Public object element=null;
    Public DLNode prev=null;
    Public DLNode next=null;

    Public DLNode(Object e, DLNode p, DLnode n){
        Element=e;
        Prev=p;
        Next=n;
    }
}
```


Chapter 6 Lists

- 6.1 Collections
- 6.2 List ADT
- 6.3 A simple array-based implementation of a list
- 6.4 A node-based implementation of a list — doubly linked list
- 6.5 Java list classes
- 6.6 Iterators

6.1 Collections

- A collection represents a group of objects (elements).
  - Allows or does not allow duplicate elements
  - Is ordered or unordered.
- Collection hierarchy



6.2 List ADT

- List, also known as sequence, is an ordered collection. Normally, it can index into the middle of a sequence and it also provides update methods for adding and removing elements by their positions.
- Stacks and queues are special kinds of Lists.

6.2 List ADT

- Fundamental methods
  - Add an element
    - add(int index, Object element)
    - addFirst(Object element)
    - addLast(Object element)
  - Remove an element
    - Object remove(int index)
    - boolean remove(Object element)
    - Object removeFirst()
    - Object removeLast()

6.2 List ADT

- Get an element
  - Object get(int index)
  - Object getFirst ()
  - Object getLast()
- Set an element
  - set(int index, Object element)
- Find the index of an element
  - int indexOf(Object element)
  - Int lastIndexOf (Object element)

6.2 List ADT

- Other methods
  - clear()
  - boolean contains(Object element)
  - int size()

6.3 A Simple Array-Based Implementation of a List

```
public class ArrayBasedList{
        private  capacity;
        public Object s[];        //Array used to implement the List
        private int num = 0;      //number of the elements stored

        public ArrayBasedList(int N){
                capacity = N;
                s = new Object[N];
        }

        //declaration of methods
        ….
}
```


6.3 A Simple Array-Based Implementation of a List

- Insert an element

```
public void add(int index, Object element){
        if(index>=this.capacity)
                return;
        for(int i=num; i>index; i--)
                S[i]=S[i-1];
        S[i-1] = element;
        num++;
        return;
}
```

6.3 A Simple Array-Based Implementation of a List

- Remove an element at a specified position

```
public void removeElementAt(int index){
        if(index>=this.capacity)
                return;
        for(int i=index; i<num; i++)
                S[i]=S[i+1];
        S[num-1] = null;
        num--;
        return;
}
```


6.3 A Simple Array-Based Implementation of a List

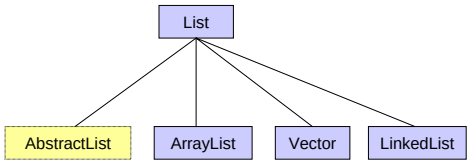
- Running time analysis

Method	Time
add(int index, Object O)	O(n)
addFirst(Object O)	O(n)
addLast(Object O)	O(1)
remove(int index)	O(n)
remove(Object O)	O(n)
removeFirst()	O(n)
removeLast()	O(1)
get(int index)	O(1)
getFirst()	O(1)
getLast()	O(1)
set(int index, Object O)	O(1)
indexOf(Object O)	O(n)

6.4 A Node-based Implementation of a List — Doubly Linked List

Method	Time(Array-based)	Time(Doubly Linked List)
add(int index, Object O)	O(n)	O(n)
addFirst(Object O)	O(n)	O(1)
addLast(Object O)	O(1)	O(1)
remove(int index)	O(n)	O(n)
remove(Object O)	O(n)	O(n)
removeFirst()	O(n)	O(1)
removeLast()	O(1)	O(1)
get(int index)	O(1)	O(n)
getFirst()	O(1)	O(1)
getLast()	O(1)	O(1)
set(int index, Object O)	O(1)	O(n)
indexOf(Object O)	O(n)	O(n)

6.5 Java List Classes



6.5 Java List Classes

- java.util.Vector class
  - Implements a growable array of objects.
  - It has an integer *CapacityIncrement* parameter to determine how the underlying extendable array grows
    - If *CapacityIncrement* =0 (defalut), the array doubles when it grows.
    - If *CapacityIncrement* =k (k>0), the array adds k cells when it grows.

6.5 Java List Classes

- Capacity related constructor and methods
  - Vector(int initialCapacity);
  - Vector(int initialCapacity, int capacityIncrement)
  - void ensureCapacity(int minCapacity)
  - void setSize(int newSize)
- Is Synchronized.

6.5 Java List Classes

- java.util.ArrayList class
  - This class is roughly equivalent to Vector, except that it is unsynchronized.
  - If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.

6.6 Iterators

- A typical computation on collections is to march through its elements in order, one at a time, to look for a specific element.
- A class that implements Java's Iterator interface provides three methods:
  - public boolean hasNext();
  - public Object next();
  - public void remove();

6.6 Iterators

- Java.util.Iterator provides a generic mechanism for scanning through a container, ADTs storing collections of objects in Java support a method iterator() that returns an iterator of the elements in the collection.

6.6 Iterators

- Example

```
public static void printVector(java.util.Vector, V){
    java.util.Iterator iter = V.iterator();
    while(iter.hasNext())
        System.out.println(iter.next());
    }
}
```

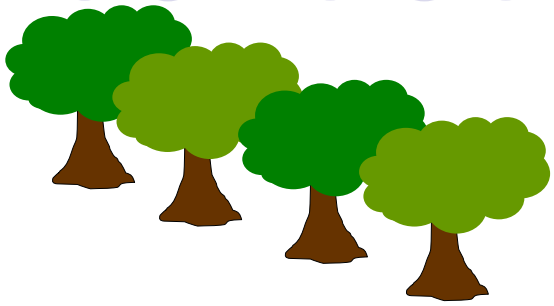

Chapter 7 Trees

- 7.1 Tree Terminology and Basic Properties
- 7.2 Tree ADT
- 7.3 Data Structures for Representing Trees
  - 7.3.1 A Vector-Based Structure for Binary Trees
  - 7.3.2 A Linked Structure for Binary Trees
- 7.4 Traversals of binary trees
  - 7.4.1 Pre-order traversal
  - 7.4.2 In-order traversal
  - 7.4.2 Post-order traversal
- 7.5 Binary Search Trees

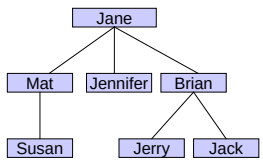
Chapter 7 Trees

- 7.6 Heaps
  - 7.6.1 The heap implementation of a priority tree
  - 7.6.2 Creating a heap
  - 7.6.3 Running time analysis

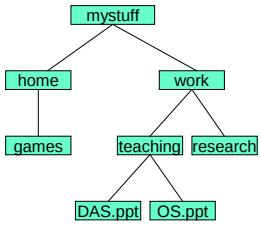
Chapter 7 Trees



Chapter 7 Trees



Jane's children and grandchildren



File organization in a computer

Chapter 7 Trees

- Tree is one of the most important nonlinear data structures in computing.
- It allows us to implement faster algorithms(compared with algorithms using linear data structures).

7.1 Tree Terminology and Basic Properties

- **Definition**: A Tree is a set of nodes storing elements in a parent-child relationship with the following properties:
  - It has a special node called **root**.
  - Each node different from the **root** has a **parent** node.
- **Terms**
  - **Parent** — the parent of a node is the node linked above it.

7.1 Tree Terminology and Basic Properties

- **Sibling** — Two nodes are siblings if they have the same parent.
- **Ancestor**
- **Descendant**
- **Leaf** — a node which has no child.
- **Subtree** — any node and its descendants form a subtree of the original tree.
- **Path of two nodes** — a path that begins at the starting node and goes from node to node along the edges that join them until the ending node.
- **Length of a path** — the number of the edges that compose it.

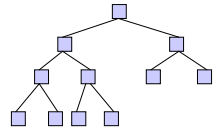
7.1 Tree Terminology and Basic Properties

- **Depth of a node** — the length of the path between the root and the node.
- **Height of a tree** — the maximum depth of a leaf node.
- Tree Types
  - Binary tree — each node has at most two children
  - n-ary tree — each node has at most n children
  - General tree — each node can have an arbitrary number of children.

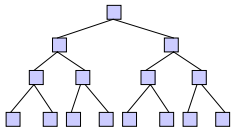
7.1 Tree Terminology and Basic Properties

- More terms
  - A binary tree is full if every non-leaf node in the tree has exactly two children
  - A binary tree is complete if every level except the deepest contains as many nodes as possible, and all the nodes at the deepest level are as far left as possible.
    - A complete binary tree of depth n has
      - Maximum number of leaves: 2^n
      - Maximum number of total nodes: 2^{n+1} -1.

7.1 Tree Terminology and Basic Properties



A full binary tree



A complete binary tree

7.1 Tree Terminology and Basic Properties

- Properties of Binary Trees
  - Let T be a full binary tree with *n* nodes, and let *h* denote the height of T, then T has the following properties:

	At least	At most
The number of leaf nodes	*h+1*	2^h
The number of non-leaf nodes	*h*	$2^h - 1$
The total number of nodes	*2h+1*	$2^{h+1} - 1$
The height of the tree *h*	$\log(n+1) - 1$	*(n-1)/2*

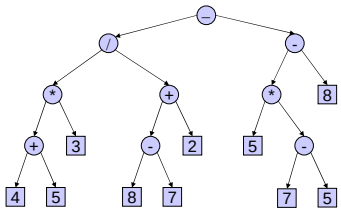
7.1 Tree Terminology and Basic Properties

Example 7.1: An arithmetic expression can be represented by a tree whose leaf nodes are associated with variables or constants, and whose non-leaf nodes are associated with one of the operators +, -, x and /.

- Such an arithmetic expression tree is normally a full binary tree, since each of the operators +, -, *, and / take exactly two operands.

7.1 Tree Terminology and Basic Properties

Given a expression (((4+5)*3)/((8-7)+2))-((5*(7-5))-8), Its expression tree is as follows:



7.2 Tree ADT

- Tree nodes
  - The tree ADT stores elements at nodes which are defined relative to neighbouring nodes.
  - Themselves are ADTs which support the method
    - Object element(): return the object at this node.
- Tree methods
  - Accessor methods
    - TNode root()
    - TNode parent(TNode node)
    - Iterator children(TNode node)

7.2 Tree ADT

- Query methods
  - boolean isLeaf(TNode node)
  - boolean isRoot (TNode node)
- Generic methods
  - Int size();
  - Iterator elements();
  - Iterator nodes();
  - swapElements(TNode node1, TNode node2)
  - TNode replaceElement(TNode node, Object element)

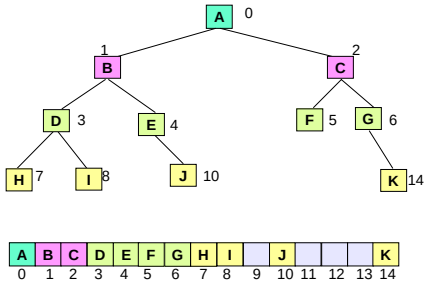
7.2 Tree ADT

- Binary tree methods. Three additional accessor methods are supported
  - TNode leftChild(TNode node)
  - TNode rightChild(TNode node)
  - TNode sibling(TNode node)

7.3 Data Structures for Representing Trees

- 7.3.1 A Vector-Based Structure for Binary Trees
  - Level numbering function *p* of nodes in a binary tree T is defined as follows:
    - If *v* is the root of T, then *p(v)=0*;
    - if *v* is the left child of node u, then *p(v)=2p(u)+1*
    - if *v* is the right child of node u, then *p(v)=2p(u)+2*
  - Represent a binary tree T by means of a Vector S : node *v* of T is associated with the element of S at position *p(v)*.

7.3.1 A Vector-Based Structure for Binary Trees



7.3.1 A Vector-Based Structure for Binary Trees

- Running times of methods of a binary tree implemented with a vector

Methods	Time
Nodes(), elements()	O(n)
swapElements(), replaceElement()	O(1)
Root(), parent(), children()	O(1)
leftChild(), rightChild(), sibling()	O(1)
isLeaf(), isRoot()	O(1)

7.3.1 A Vector-Based Structure for Binary Trees

- Running times of methods of a binary tree implemented with a vector

Methods	Time
Nodes(), elements()	O(n)
swapElements(), replaceElement()	O(1)
Root(), parent(), children()	O(1)
leftChild(), rightChild(), sibling()	O(1)
isLeaf(), isRoot()	O(1)

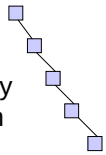
7.3.1 A Vector-Based Structure for Binary Trees

- Pros: Fast and easy
- Cons: Can be very space inefficient if the height of the tree is large

7.3.1 A Vector-Based Structure for Binary Trees

- **Example 7.1**: If the number of nodes in a binary tree is 5, what is the size of the Vector in the worst case?

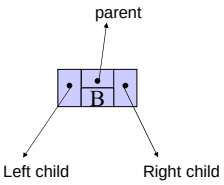
  $1+2+4+8+16 = 31 = 2^{5-1}$



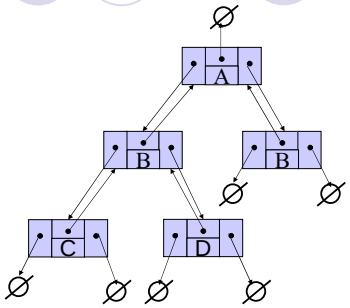
- If the number of nodes in a binary tree is n, the size of the Vector in the worst case is 2^n-1

7.3.2 A Linked Structure for Binary Trees

- We represent each node of a binary tree by an object which stores
  - Element
  - References to its parent and children nodes



7.3.2 A Linked Structure for Binary Trees



7.3.2 A Linked Structure for Binary Trees

- BTNode class declaration

```
public class BTNode{
    private Object element;
    public BTNode left, right, parent;

    public BTNode(){}

    public BTNode(Object O, BTNode l, BTNode r, BTNode p){
        this.element = O; this.left =l;
        this.right = r; this.parent =p;
    }

    public Object element(){ return this.element;}
}
```


7.3.2 A Linked Structure for Binary Trees

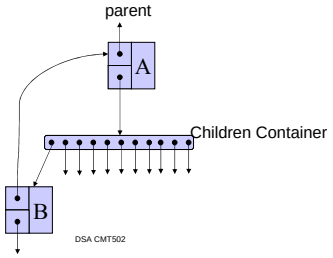
- Running times of methods of a binary tree implemented with a linked structure

Methods	Time
Nodes(), elements()	O(n)
swapElements(), replaceElement()	O(1)
Root(), parent(), children()	O(1)
leftChild(), rightChild(), sibling()	O(1)
isLeaf(), isRoot()	O(1)

- Space usage is O(n) for an n-node binary tree.

7.3.3 A Linked Structure for General Trees

- A container (for example, a list or vector) to store the children of a node in a general tree.



7.4 Traversals of Binary Trees

- A traversal of a tree is a systematic way of accessing or "visiting" all the nodes in the tree.
- There are three basic traversal schemes:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

7.4.1 Pre-Order Traversal

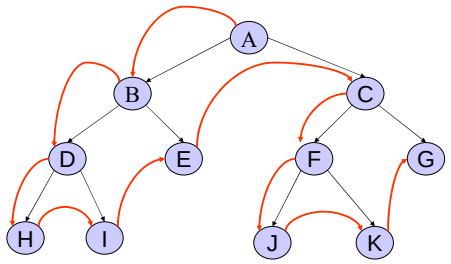
- A pre-order traversal has three steps for a nonempty tree:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.

7.4.1 Pre-Order Traversal

- Algorithm binaryPreorder(T, v)

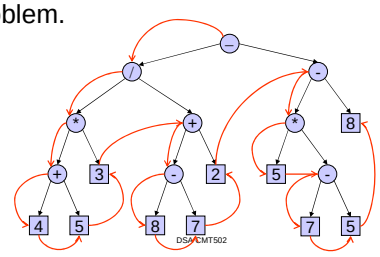
```
Input: binary tree T, node v
Output: none

binaryPreorder(T, v){
    Perform the "visit" action for node v
    If(!T.isLeaf(V)){
        binaryPreorder(T, T.leftChild(v))
            //recursively traverse left subtree
        binaryPreorder(T, T.rightChild(v))
            //recursively traverse right  subtree
    }
}
```


7.4.1 Pre-Order Traversal



7.4.1 Pre-Order Traversal

- Using pre-order traversal of a binary tree to solve the expression evaluation problem.



7.4.1 Pre-Order Traversal

Input: binary tree T, node v
Output: result of the sub expression

```
preorderEvaluateExpression(T, v){
    if(T.isLeaf(v))
        return v.element()
    operator = v.element()
    operand1 = preorderEvaluateExpression (T, T.leftChild(v))
    operand2 = preorderEvaluateExpression (T, T.rightChild(v))
    compute the expression and get the result
    return result
}
```


7.4.2 In-Order Traversal

- An in-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

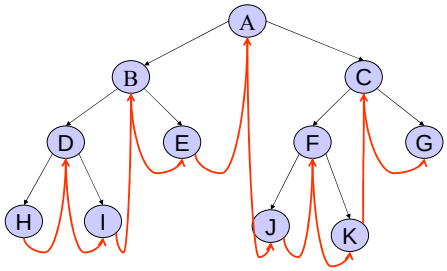
7.4.2 In-Order Traversal

- Algorithm binaryInorder(T, v)

Input: binary tree T, node v
Output: none

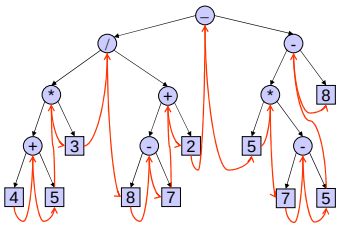
```
binaryInorder(T, v){
    If(!T.isLeaf(v))
        binaryInorder(T, T.leftChild(v))  //recursively traverse left
            subtree
    Perform the "visit" action for node v
    If(!T.isLeaf(v))
        binaryInorder(T, T.rightChild(v))  //recursively traverse right
            subtree

}
```


7.4.2 In-Order Traversal



7.4.2 In-Order Traversal

- Using in-order traversal of a binary tree to solve the expression evaluation problem.



7.4.2 In-Order Traversal

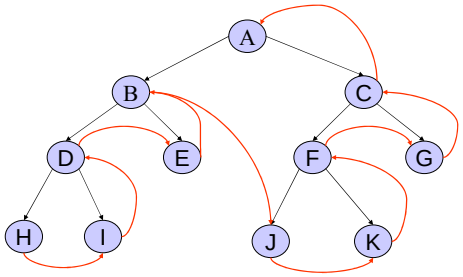
Input: binary tree T, node v
Output: result of the sub expression

```
inorderEvaluateExpression(T, v){
    if(T.isLeaf(v))
        return v.element()
    operand1 = inorderEvaluateExpression (T, T.leftChild(v))
    operator = v.element()
    operand2 = inorderEvaluateExpression (T, T.rightChild(v))
    compute the expression and get the result
    return result
}
```

7.4.3 Post-Order Traversal

- A post-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.

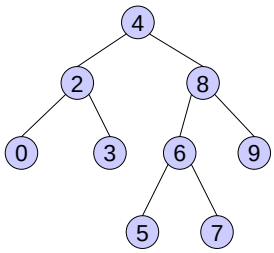
7.4.3 Post-Order Traversal



7.5 Binary Search Trees

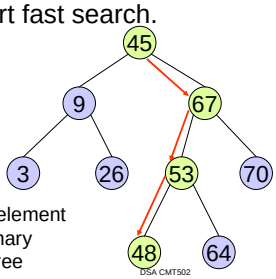
- In a binary search tree, each non-leaf node v stores an element e such that
  - the elements stored in the left subtree of v are less than or equal to e.
  - the elements stored in the right subtree of v are greater than e.

7.5 Binary Search Trees



7.5 Binary Search Trees

- Advantage of using binary search trees — support fast search.



Find an element in the binary search tree

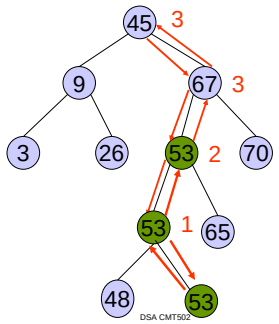
7.5 Binary Search Trees

- Counting the occurrences of an element in a binary search tree.

Input: binary search tree T, node v, element e
Output: the number of the occurrences of element e in a subtree with v as its root node.

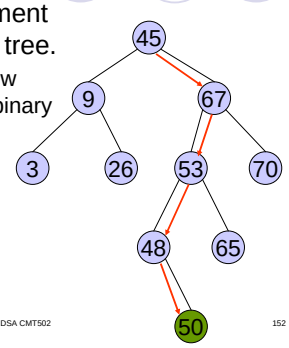
```
occurrencesCount(T, v, e){
    int ret=0;
    if(T.isLeaf(v))
        if(v.element()==e)  ret=1;
    else {
        if(v.element()==e)
            ret= 1+occurrenceCount(T, T.leftChild(v), e);
        else if(v.element()>=e)
            ret = occurrenceCount(T, T.leftChild(v), e);
        else   ret=occurrenceCount(T, T.rightChild(v), e);
    }
    return ret;
}
```

7.5 Binary Search Trees



7.5 Binary Search Trees

- Adding a new element to a binary search tree.
  - Example: add a new element 50 to the binary search tree.



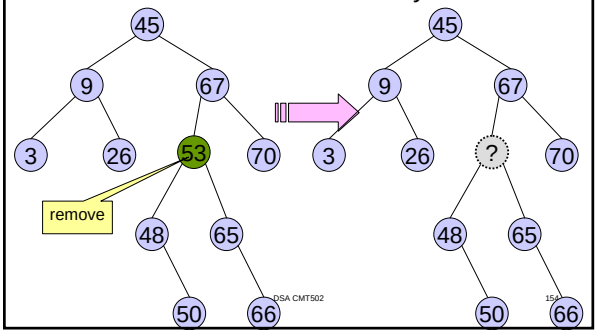
7.5 Binary Search Trees

- Adding a new element to a binary search tree.

```
Input: binary search tree T, node v, element e
Output:
add(T, v, e){
    if(T.isLeaf(v)){
        if(v.element()>=e)
            add element e as v's left child
        else
            add element e as v's right child
    } else {
        if(v.element()>=e)
            add(T, T.leftChild(v), e)
        else
            add(T, T.rightChild(v), e)
    }
}
```


7.5 Binary Search Trees

- Remove an element from a binary search tree.

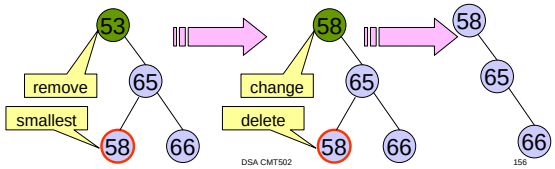


7.5 Binary Search Trees

- Input: binary search tree, node v, element e
- Output: boolean
- Algorithm *remove(T, v, e)*: removes a copy of element from a subtree whose root is node v. It returns true if a copy of the element has been removed, otherwise return false.

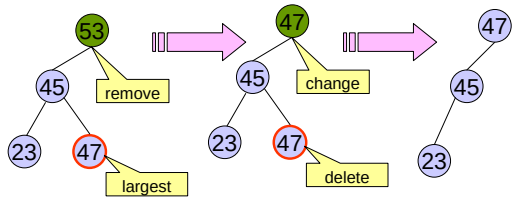
7.5 Binary Search Trees

- In algorithm *remove(T, v, e)*,
  - If node v is a leaf node and v stores element e.
    - Delete v node
  - If node v only has right child and v stores element e.
    - Find the smallest element *s* in v's right subtree, replace v's data with s and remove the node that originally stored the element s.



7.5 Binary Search Trees

- If node v has left child and v stores element e.
  - Find the largest element *l* in v's left subtree, replace v's data with *l* and remove the node that originally stored the element s.



Input: Binary search tree T, node v, element e
Output: true if the node storing element e has been removed, otherwise false

```
remove(T, v, e){
    if(v==null) return false;
    if(v.element==e){
        if(T.isLeaf(v)){
            if(T.leftChild(T.parent(v))==v) // v is the left child of its parent
                (T.parent(v)).left = null;
            else  // v is the right child of its parent
                (T.parent(v)).right = null;
        } else {
            BTNode s = null;
            if(T.leftChild(v)!=null)    s=maxNode(T, T.leftChild(v));
            else   s=minNode(T, T.rightChild(v));
            T.replaceElement(v, s.element());
            remove(T, s, s.element());
        }
        return true
```

To be continued


```
    } else {
        if(T.isLeaf(v)) return false
        else if (v.element()> =  e)
            return remove(T, T.leftChild(v), e)
        else
            return remove(T.T.rightChild(v), e)
    }
}
```


Input: Binary search tree T, node v
Output: return the largest node in the subtree rooted at v

```
maxNode(T, v){
    if(T.rightChild(v)==null) return v
    else return maxNode(T, v.rightChild(v))
}
```

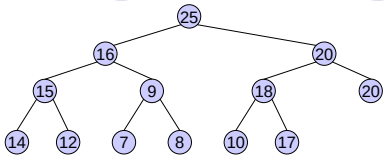
Input: Binary search tree T, node v
Output: return the smallest node in the subtree rooted at v

```
minNode(T, v){
    if(T.leftChild(v)==null) return v
    else return minNode(T, v.leftChild(v))
}
```


7.6 Heaps

- A heap is a binary tree that satisfies two additional properties:
  - The element contained by each non-leaf node is greater than or equal to the element stored at that node's children.
  - The tree is a complete binary tree so that every level except the deepest must contain as many nodes as possible; at the deepest level, all the nodes are as far left as possible.

7.6 Heaps

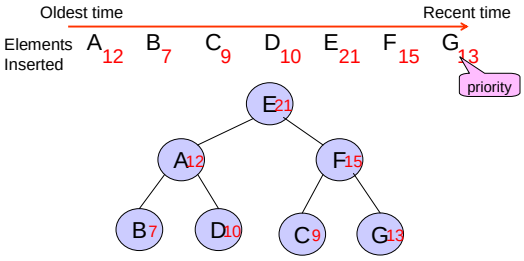


Example of a heap storing 13 elements

7.6.1 The Heap Implementation of a Priority Queue

- In the heap implementation of a priority queue,
  - each node of the heap contains one element along with the element's priority
  - The tree is maintained so that it follows the heap storage rules using the element's priority to compare nodes.

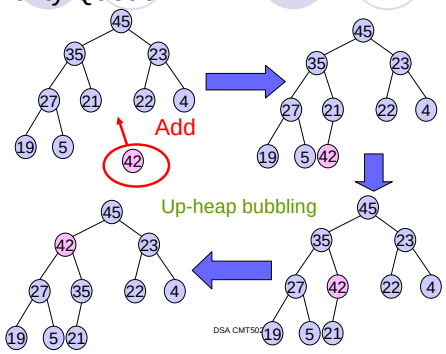
7.6.1 The Heap Implementation of a Priority Queue



7.6.1 The Heap Implementation of a Priority Queue

- Adding an element to a heap
  - Place the new element in the heap in the first available location. (This keeps the structure as a complete binary tree, but it might no longer be a heap)
  - Swap the new element with its parent if it is greater than its parent's element, until it is less than or equal to its parent's element.
  - This is called up-heap bubbling

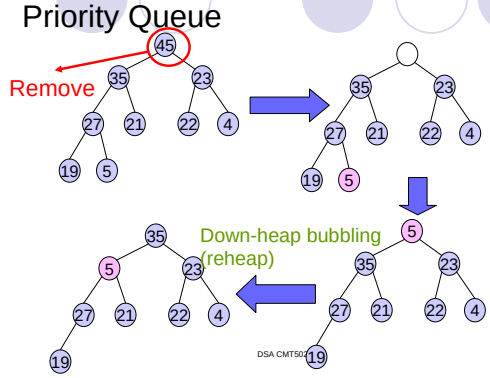
7.6.1 The Heap Implementation of a Priority Queue



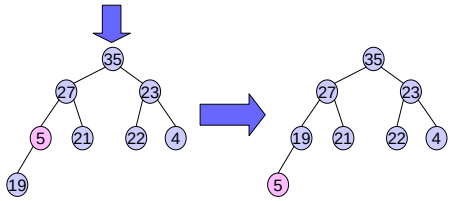
7.6.1 The Heap Implementation of a Priority Queue

- Removing the root element from a heap
  - When an element is remove from a priority queue, we must always remove the element with the highest priority
    1. Remove the root node in the heap
    2. Move the last element v in the last level to the root.
    3. If any of v's child elements are larger than or equal to v, swap node v with its largest child. Repeat this step until all v's children are smaller than v. This is called down-heap bubbling or reheap.

7.6.1 The Heap Implementation of a Priority Queue

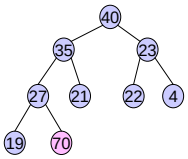


7.6.1 The Heap Implementation of a Priority Queue



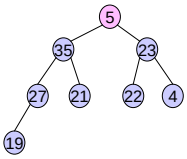
7.6.1 The Heap Implementation of a Priority Queue

- Upheap algorithm: transform a semiheap, in which except for the last leaf, the elements are ordered as they are in a heap, into a heap.



7.6.1 The Heap Implementation of a Priority Queue

- Downheap algorithm: transforms a semiheap, in which, except for the root, the elements are ordered as they are in a heap, into a heap.



7.6.1 The Heap Implementation of a Priority Queue

upheap Algorithm

```
Input: Array H, int lastIndex
Output:
upheap(H, lastIndex){
    if H[lastIndex] is the root,  return
    parentIndex=(lastIndex-1)/2

    if (H[lastIndex] >  H[parentIndex]){
        temp = H[lastIndex]
        H[lastIndex] = H[parentIndex]
        H[parentIndex] = temp
        upHeap(H, parentIndex)
    }
}
```


7.6.1 The Heap Implementation of a Priority Queue

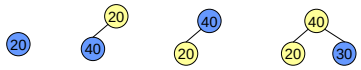
- Downheap Algorithm

```
Input: Array H, int rootIndex
Output:
downheap(H, rootIndex){
    if H[rootIndex] is a leaf,  return
    childIndex=index of the larger of the root's children.

    if (H[rootIndex] <  H[childIndex]){
        temp = H[rootIndex]
        H[rootIndex] = H[childIndex]
        H[childIndex] = temp
        downHeap(H, childIndex)
    }
}
```

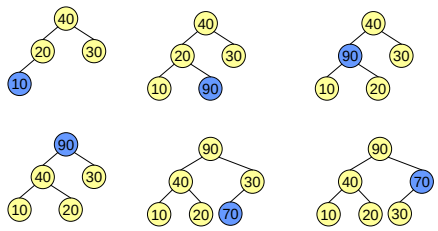

7.6.2 Creating a Heap

- Using add: create a heap from a collection of objects by using the add method to add each object to an initially empty heap.

Example: adding 20, 40, 30, 10, 90, and 70 to a heap



7.6.2 Creating a Heap



7.6.2 Creating a Heap

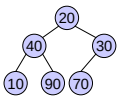
- Using downheap:
  - Place the entries for the heap into an array beginning at index 0. This array actually represents a complete tree.
  - Starting at the end of the array, ignoring the items which are leaves in the complete tree, and moving towards the beginning of the array, the next item we encounter is the root of a semiheap within the tree. Apply downheap to this semiheap. Continue in this manner.

7.6.2 Creating a Heap

Example: adding 20, 40, 30, 10, 90, and 70 to a heap

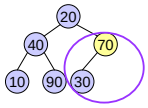
(a) The array of all entries and the complete tree that the array represents.

0	1	2	3	4	5
20	40	30	10	90	70



(b) After downheap(A, 3)

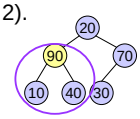
0	1	2	3	4	5
20	40	70	10	90	30



7.6.2 Creating a Heap

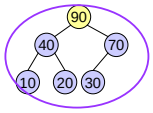
(c) After downheap(A, 2).

0	1	2	3	4	5
20	90	70	10	40	30



(d) After downheap(A, 1)

0	1	2	3	4	5
90	40	70	10	20	30



7.6.2 Creating a Heap

- Heap construction using upheap method
  - Implementing upheap in Java

    In the following algorithm, input array A starts at index 0 and A[0…lastIndex-1] represents a heap. Now add element A[lastIndex] into the heap. This method transforms the semiheap A[0…lastIndex] into a heap.


```
private static void upheap(int[] A, int lastIndex){
    if(lastIndex==0)
        return;

    int parentIndex = (lastIndex-1)/2;
    if(A[lastIndex]>A[parentIndex]){
        int temp=A[lastIndex];
        A[lastIndex] = A[parentIndex];
        A[parentIndex]=temp;
        upheap(A, parentIndex);
    }
}
```

7.6.2 Creating a Heap

- Implementing heap construction by using the upheap method

```
public static void createHeap(int[] A){
    for(int i=1; i<A.length; i++)
        upheap(A, i);
}
```


7.6.2 Creating a Heap

- Heap construction using the downheap method
  - Implementing downheap in Java

    In the following algorithm, sub-array A[rootIndex…end] represents a semiheap, in which, except for the root element rootIndex, all other elements are ordered as they are in a heap. This method transforms the semiheap into a heap.


```
private static void downheap(int[] A, int rootIndex, int end){
        if(rootIndex > (end-1)/2)  //entry rootIndex is a leaf
            return;
    int childIndex = 2*rootIndex+1;
    if(A[childIndex] < A[childIndex+1])
        childIndex += 1;
    if(A[rootIndex]<A[childIndex]){
        int temp=A[rootIndex];
        A[rootIndex] = A[childIndex];
        A[childIndex]=temp;
        downheap(A, childIndex,end);
    }
}
```


7.6.2 Creating a Heap

- Implementing Heap construction using the downheap method

```
Public static void createHeap(int[] A){
    for(int i=(A.length-1)/2    ; i>0; i--)
        downheap(A, A.length-1);
}
```


7.6.3 Running time analysis

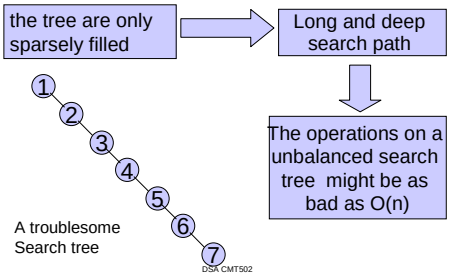
- The adding(upheap) and removing(downheap) operators can be performed in O(logn) time, where n is the number of elements. This is based on the following:
  - Since T is complete, the height of a heap is O(logn)
  - In the worst case, the upwards or downwards swapping take time proportional to the height of the heap.
  - All other operations takes constant time
- Both of heap construction methods (upheap and downHeap) can be performed in O(nlogn) time.

Chapter 8 Balanced Search Trees

- 8.1 AVL Trees
- 8.2 2-3 Trees
- 8.3 B-Trees

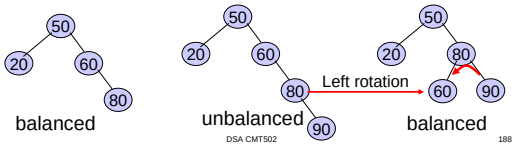
Chapter 8 Balanced Search Trees

- The problem of unbalanced trees



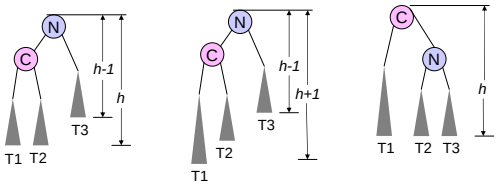
8.1 AVL Trees

- An AVL tree is a binary search tree that rearranges its nodes whenever it becomes unbalanced.
- A node is balanced if its two subtrees differ in height by no more than 1.



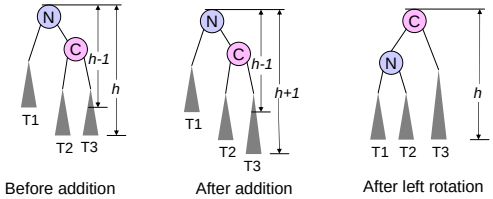
8.1 AVL Trees

- Single rotations
  - Right rotation — when the addition occurs in the left subtree of node N's left child.



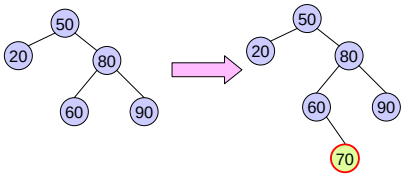
8.1 AVL Trees

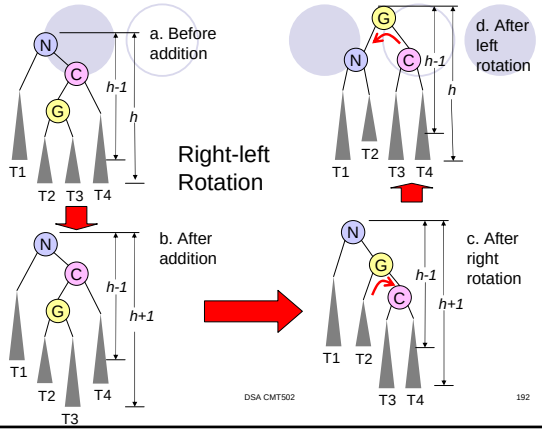
- Left rotation — when the addition occurs in the right subtree of node N's right child.



8.1 AVL Trees

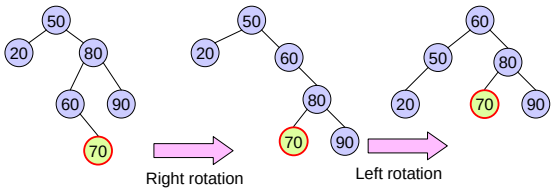
- Double rotations
  - Right-left rotation — when the addition occurs in the left subtree of node N's right child.





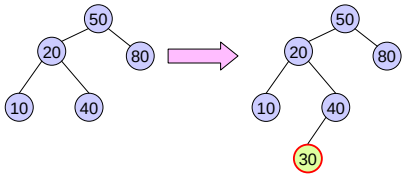
8.1 AVL Trees

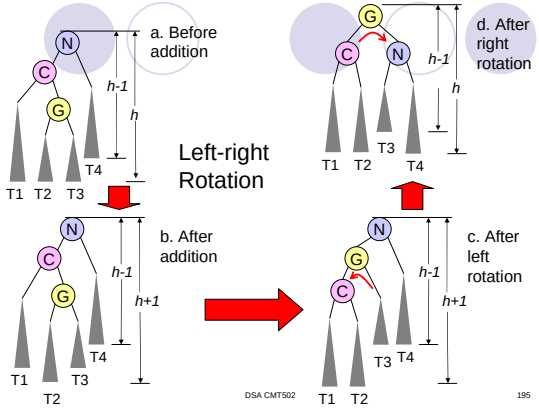
Example of right-left rotation



8.1 AVL Trees

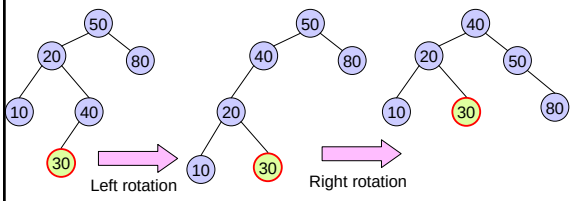
- Left-right rotation — when the addition occurs in the right subtree of node N's left child.





8.1 AVL Trees

Example of left-right rotation



8.2 2-3 Trees

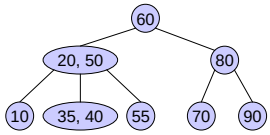
- A 2-3 tree is a general search tree which follows the following rules
  - All its non-leaf nodes must be either 2-node or 3-node.
  - All its leaf nodes occur on the same level.
- 2-node and 3-node
  - A 2-node contains one data element and has two children. The data element is greater than any data in the node's left subtree and less than any data in the right subtree.

8.2 2-3 Trees

- A 3-node contains two data elements, s and l, and has three children. Assume s<l.
  - Data elements that are less than s occur in the node's left subtree
  - Data elements that are larger than s and less than l occur in the node's middle subtree
  - Data elements that are greater than l occur in the node's right subtree
- A 2-3 tree is completely balanced.

8.2 2-3 Trees

- An example



8.2 2-3 Trees

- Searching a 2-3 tree

Input: 2-3 Tree T, node v element e
Output: boolean

```
search23Tree(T, v, e){
    if (v contains e) return true
    else if(v is a leaf node)  return false
    Find the index number i by comparing e with the elements stored in v
    return search23Tree(T, v's ith child, e)
}
```

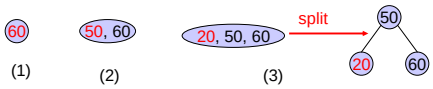
- It runs in O(logn) time

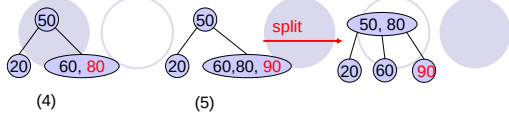
8.2 2-3 Trees

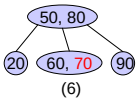
- Adding elements to a 2-3 tree

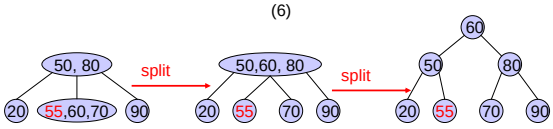
Example 8.1 Adding the following elements to an empty 2-3 tree

60, 50, 20, 80, 90, 70, 55, 10, 40, and 35

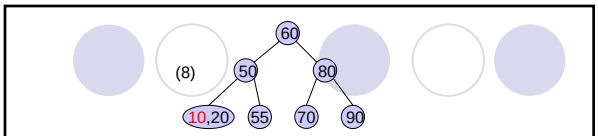


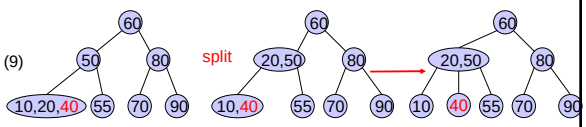


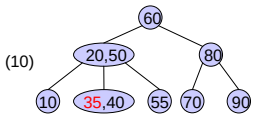




(7)

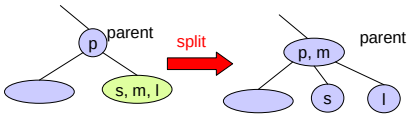






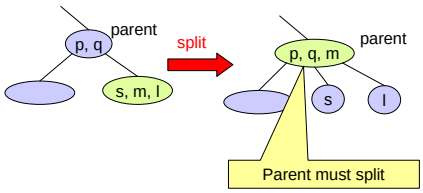
8.2 2-3 Trees

- Splitting nodes during addition
  - Split a leaf node to accommodate a new element
    - If the leaf's parent contains one data element.



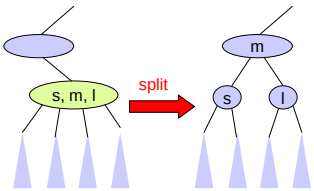
8.2 2-3 Trees

- If the leaf's parent contains two data elements.



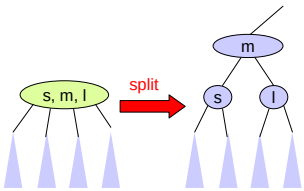
8.2 2-3 Trees

- Split a non-leaf node to accommodate a new element



8.2 2-3 Trees

- Split a root node to accommodate a new element



8.3 B-Trees

- A B-tree is not a binary tree. It nodes can have many more than two children.
- Each node in a B-tree might contain more than one element.

8.3 B-Trees

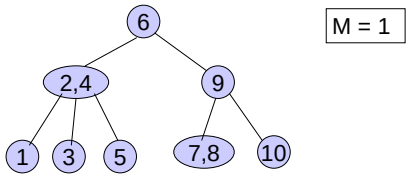
- B-tree rules
  - Rules for the element in a B-Tree Node: There is a positive constant integer M which determines the number of elements stored in a single node.
    1. The root can have as few as one element (or even no elements if it also has no children)
    2. All nodes other than the root have at least M elements and at most 2M elements
    3. The elements of each B-tree node are stored in a particular container, sorted in an ascending order.

8.3 B-Trees

- Rules for the subtrees below a B-Tree Node:
  1. The number of subtrees below a non-leaf node is always one more than the number of elements in the node.
  2. The elements in each subtree are organized in such a way:

     For any non-leaf node, an element at index i is greater than all the elements in subtree number i of the node, and less than all the elements in subtree number i+1 of the node
- Rule for balancing the tree: Every leaf in a B-tree has the same depth.

8.3 B-Trees



A Example of B-Tree

8.3 B-Trees

- Search an element in a B-tree
  - Assume in the B-Tree nodes ADT BaTNode, two additional methods provided:
    - int getElementNum (): returns the number of elements stored in the node.
    - int getIndex(Object e): returns an index number i such that
      - i=0, if e is less than the first element stored in the node
      - 0<i<getElementNum()-1, if e is larger than the (i-1)th element and less than the ith element stored in the node
      - i=getElementNum()-1, if e is larger than the last element stored in the node

8.3 B-Trees

- boolean inNode (Object e): returns true if the element e is one of the elements stored in the node, otherwise false.
- Also assume in the B-Tree ADT, the following method is provided:
  - BaTNode getChild(BaTNode v, int i) returns node v's ith child node.

8.3 B-Trees

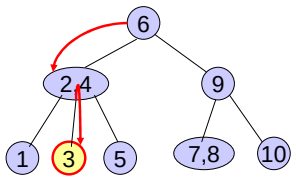
- Algorithm for searching a B-tree

Input B-Tree T, BaTNode v and element e
Output: a Boolean value

```
searchBaTree(T, v, e){
    if(v.inNode(e)) return true
    else if (T.isLeaf(v)) return false
    i = v.getIndex(e);
    return searchBaTree(T, T.getChild(v, i), e)
}
```

- Running time is O(logn)

8.3 B-Trees



An Example of a B-Tree Search

Chapter 9 Sorting

Chapter 9 Sorting

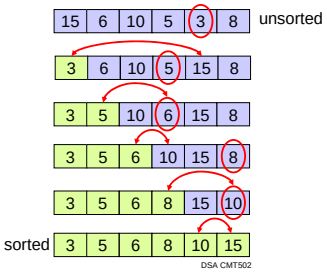
- Arranging things into ascending or descending order is called sorting.

9.1 Selection Sort

- In terms of an array A, the selection sort finds the smallest element in the array and exchanges it with A[0]. Then, ignoring A[0], the sort finds the next smallest and swaps it with A[1] and so on.

9.1 Selection Sort

- An example of selection sort



9.1.1 Iterative Selection Sort

- Iterative selection sort algorithm

```
//Sort the first n elements of an array
Input: array A, int n
Output:
selectionSort1(A, n){
    for(index = 0; index<n-1; index++){
        indexOfNextSmallest = the index of the smallest value
        among A[index], A[index+1], …A[n-1]
        interchange the value of A[index] and A[indexOfNextSmallest]
    }
}
```


9.1.1 Iterative Selection Sort

- Implementing iterative selection sort in Java

```
public static void selectionSort1(int[] A, int n){
    for(int i = 0; i<n-1; i++){
        int indexOfNextSmallest = i;
        int smallest = A[i];
        for(int j=i+1; j<n; j++){
            if(smallest>A[j]){
                indexOfNextSmallest=j;
                smallest=A[j];
            }
        }
        A[indexOfNextSmallest]=A[i];
        A[i]=smallest;
    }
}
```


9.1.2 Recursive Selection Sort

- Recursive selection sort algorithm

```
//Sort the array elements A[first] through A[last] recursively
Input: array A, int first, int last
Output:
selectionSort2(A, first, last){
    if(first<last){
        indexOfNextSmallest = the index of the smallest value among
            A[first], A[first+1], …A[last]
        interchange the value of A[first] and A[indexOfNextSmallest]
        selectionSort2(A, first+1, last);
    }
}
```

9.1.2 Recursive Selection Sort

- Implementing recursive selection sort in Java

```
public static void selectionSort2(int[] A, int first, int last){
        if(first>=last) return;
        int indexOfNextSmallest = first;
        int smallest = A[first];
        for(int i=first+1; i<=last; i++){
            if(A[i]<smallest){
                smallest=A[i];
                indexOfNextSmallest=i;
            }
        }
        A[indexOfNextSmallest]=A[first];
        A[first]=smallest;
        selectionSort2(A, first+1, last);
}
```


9.1.3 Running Time Analysis

- Selection sort is $O(n^2)$ regardless of the initial order of the elements in an array.

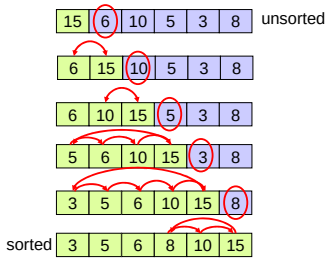
9.2 Insertion Sort

- An insertion sort of an array partitions the array into two parts.
  - One part is sorted and initially contains just the first element in the array.
  - The second part contains the remaining elements.

  The sort inserts one by one the elements in the unsorted part of the array into their proper location within the sorted part of the array.

9.2 Insertion Sort

- An example of selection sort



9.2 Insertion Sort

- Recursive insertion sort

```
//Sort the array elements A[first] through A[last] recursively
Input: array A, int first, int last
Output:

insertionSort(A, first, last){
  if (first<last){
     insertionSort(A, first, last-1)
     insert the last element A[last] into its correct
    sorted position within the rest of the array
  }
}
```


9.2 Insertion Sort

- Implementing recursive insertion sort in Java

```
public static void insertionSort(int[] A, int first, int last){
    if(first<last){
        insertionSort(A, first, last-1);
        insertInOrder(A[last], A, first, last-1);
    }
}
```

9.2 Insertion Sort

```
private static void insertInOrder(int element, int[] A,
                                  int first, int last){
    if(element>=A[last])
        A[last+1]=element;
    else if(first<last){
        A[last+1]=A[last];
        insertInOrder(element, A, first, last-1);
    } else {
        A[last+1]=A[last];
        A[last]=element;
    }
}
```


9.2 Insertion Sort

- Running time analysis
  - Insertion sort is at best O(n) and at worst O(n^2).
  - The closer an array is to sorted order, the less work an insertion sort does

9.3 Shell Sort

- Shell sort is a variation of insertion sort that is faster than O(n^2).
- Observation: the more sorted an array is, the less work the method insertInOrder() needs to do.
- Shell sort adapts the insertion sort to work on a subarray of equally spaced elements.

9.3 Shell Sort

- An example of shell sort.

0	1	2	3	4	5	6	7	8	9	10	11	12
6	7	3	8	1	9	2	10	11	13	4	12	5

6……2……5

7……10

3……11

8……13

1……4

9……12

(1) An array and the subarrays formed by grouping elements whose indices are 6 apart.

9.3 Shell Sort

(2) after the subarrays are sorted.

0	1	2	3	4	5	6	7	8	9	10	11	12
2	7	3	8	1	9	5	10	11	13	4	12	6

(3) The subarrays of the array formed by grouping elements whose indices are 3 apart.

0	1	2	3	4	5	6	7	8	9	10	11	12
2	7	3	8	1	9	5	10	11	13	4	12	6

2……..8……..5……..13 …….6

7……..1……..10…… 4

3……..9……..11…… 12

9.3 Shell Sort

(4) after the subarrays are sorted.

0	1	2	3	4	5	6	7	8	9	10	11	12
2	1	3	5	4	9	6	7	11	8	10	12	13

(5) after insertion sort.

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12	13

9.3 Shell Sort

- Efficiency of shell sort
  - The shell sort is $O(n^2)$ in the worst case. If n is a power of 2, the average running time of a shell sort is $O(n^{1.5})$.
  - By adding 1 to the space between elements any time that it is even, the worst running time of a shell sort can be improved to $O(n^{1.5})$

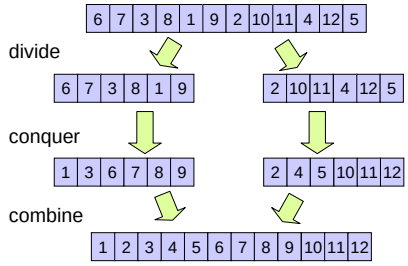
9.4 Divide-and-Conquer Sorts

- Divide-and-conquer strategy:
  - Divide a problem into pieces and conquer each piece to reach a solution.
- Divide-and-Conquer algorithms
  - Mergesort
  - Quick sort

9.4.1 Mergesort

- Mergesort paradigm
  - Divide the list of elements to be sorted into two parts of equal or almost equal size
  - Sort each part by recursive calls
  - Combine the two sorted parts into one large sorted list.

9.4.1 Mergesort



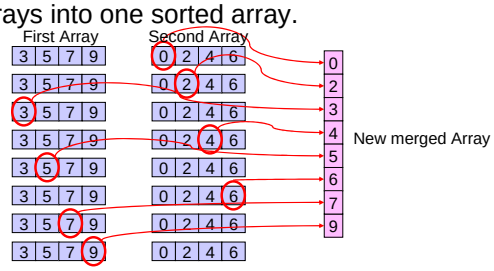
9.4.1 Mergesort

- Implementing Mergesort in Java

```
// Sort the array elements A[first] through A[first+n-1] recursively
public static void mergesort(int[] A, int first, int n){
   if(n<=1) return;
   int n1 = n/2;        //size of the first half of the array
   int n2 = n-n1;       //size of the second half of the array
   mergesort(A, first, n1);
   mergesort(A, first+n1, n2);
   merge(A, first, n1, n2);
}
```


9.4.1 Mergesort

- Merge function — Merging two sorted arrays into one sorted array.
-



```
public static void merge(int[] A, int first, int n1, int n2){
/*merge the adjacent subarrays A[first….(first+n1-1)] and A[(first+n1) …
                (first+n1+n2-1)]*/
   int[] tempA = new int[n1+n2];
   int start1 = first;
   int end1 = first+n1-1;
   int start2 = first+n1;
   int end2 = first+n1+n2-1;
   int next=0;
   while((start1<=end1) && (start2<=end2)){
          if(A[start1] <= A[start2]){
                tempA[next]=A[start1];
                start1++;
          } else {
                tempA[next]=A[start2];
                start2++;
          }
          next++;
   }
```

To be continued


```
//copy remaining elements from other subarrays to tempA.
  if(start1<end1){
          for(int i=start1; i<=end1; i++){
          tempA[next] = A[i]; next++;
          }
   } else if(start2<end2){
          for(int i=start2; i<=end2; i++){
             tempA[next] = A[i]; next++;
          }
   }

  //Copy elements from tempA to A.
   for (int i=0; i<=n1+n2; i++){
          A[start+i] = tempA[i] ;
   }
}
```


9.4.1 Mergesort

- Running time analysis
  - Assuming that $n = 2^k$

Steps	Recursive calls	Size of subarrays	Merging operations
1	2	2^{k-1}	2^k
2	2^2	2^{k-2}	2^{k-1}
3	2^3	2^{k-3}	2^{k-2}
…	…	…	…
K-1	2^{k-1}	2	2^2
K	2^k	1	2

9.4.1 Mergesort

Running time

$$2^k + 2*2^{k-1} + 2^2*2^{k-2} + ... + 2^{k-1}*2$$
$$= k*2^k$$
$$= n\log_2 n$$

- When n is not a power of 2, we can find an integer k so that $k = \lceil \log_2 n \rceil$
  Thus the running time is still O(nlogn)
- Merge steps perform the same amount of work regardless of the initial order of the array.
- Mergesort requires additional memory for merging.

9.4.1 Mergesort

- Mergesort in Java Class Library
  - java.util.Array class defines several versions of a static method sort() to sort an array into ascending order
    - Public static void sort(Object[] A)
    - Public static void sort(Object[] A, int first, int last)

9.4.2 Quicksort

- Quicksort divides an array into two pieces. These pieces are not necessarily halves of the array. It chooses one element in the array — called the pivot — and rearranges the array elements so that
  - Elements in positions before the pivot are less than or equal to the pivot
  - Elements in positions after the pivot are larger than the pivot

  The arrangement is called a partition of the array.

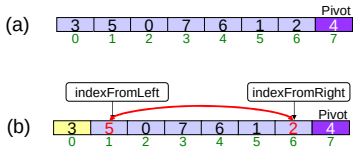
9.4.2 Quicksort

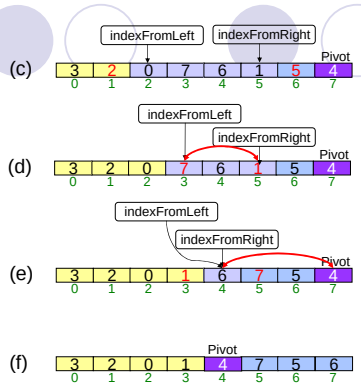
- Quicksort Algorithm

```
//Sort the array elements A[first] throught A[last]
Input: Array A, int first, int last
Output:
quicksort(A, first, last){
  if(first < last){
      Choose a pivot
      Partition the array about the pivot
      pivotIndex = index of the pivot
      quicksort(A, first, pivotIndex-1)
      quicksort(A, pivotIndex+1, last)
  }
}
```


9.4.2 Quicksort

- Creating the partition.
  - Example: assume we have chosen the last element as the pivot.





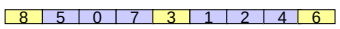
9.4.2 Quicksort

- Pivot selection
  - Ideally, the pivot should be the median value in the array, so that the two partitions have the same size — difficult to find the median value.
  - **Median-of-three pivot selection**:
    - take as pivot the median of three elements in the array: the first, the middle and the last element.

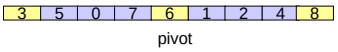
9.4.2 Quicksort

- **Median-of-three pivot selection**:

(a) The original array

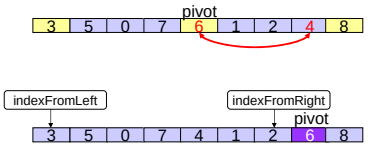


(b) The array with its first, middle and last elements sorted, and the middle element 6 is chosen as pivot.



9.4.2 Quicksort

- Adjusting the partition procedure
  - Before starting the partition procedure, swap the pivot with the next-to-last element at A[last-1]. Then the partition procedure can begin its search from the right index last-2.



9.4.2 Quicksort

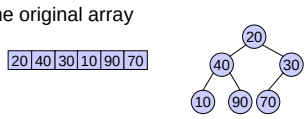
- Running time analysis
  - The choice of pivots affects the sort's efficiency. It can lead to the worst-case behaviour if the array is already sorted or nearly sorted. The worst-case running time is $O(n^2)$.
  - Quicksort is O(nlogn) in the average case.
  - Compared with mergesort, quicksort doesn't require the additional memory that mergesort needs for merging.

9.5 Heapsort

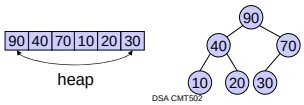
- Heapsort uses a heap to sort an array.
- Heapsort uses *downheap* instead of *upheap* to create a heap in a more efficient way.

9.5 Heapsort

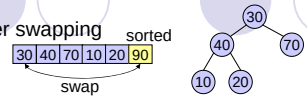
- An example of heapsort
  - (a) The original array



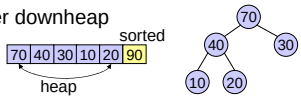
  - (b) After downheap



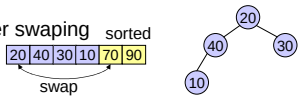
(c) After swapping



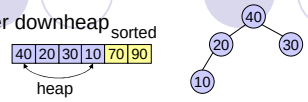
(d) After downheap



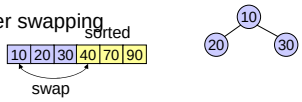
(e) After swaping



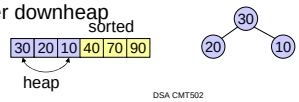
(f) After downheap



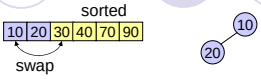
(g) After swapping



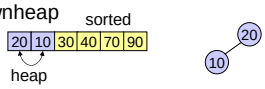
(h) After downheap



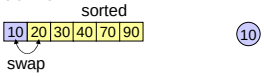
(i) After swapping



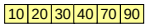
(j) After downheap



(k) After swapping



(l) Array is sorted



9.5 Heapsort

- Java code for heapsort

```
public static void heapSort(int[] A){
    createHeap(A);      // See slide 184
    swap(A, 0, A.length-1);
    hsort(A, 0, A.length-2);
}

private static void hsort(int[] A, int first, int last){
    if(first>=last) return;
    downheap(A, first, last); //See Slide 183
    swap(A, first, last);
    hsort(A, first, last-1);
}

private static void swap(int[] A, int n1, int n2){
    int temp = A[n1];
    A[n1]=A[n2];
    A[n2]=temp;
}
```


9.5 Heapsort

- Running time analysis
  - An O(nlogn) algorithm.
  - No additional memory required.

9.6 Comparing the algorithms

	Average case	Best case	Worst case	Extra array needed
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	no
Insertion sort	$O(n^2)$	$O(n\ )$	$O(n^2)$	no
Shell sort	$O(n^{1.5})$	$O(n\ )$	$O(n^{1.5})/O(n^2)$	no
mergesort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	yes
quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	no
heapsort	$O(n\log n)$	$O(n\ )$	$O(n\log n)$	no

9.6 Comparing the algorithms

A comparison of growth-rate functions as n increases

n	10	10^2	10^3	10^4	10^5	10^6
nlogn	33	664	9966	132877	1660964	19931569
$n^{1.5}$	32	10^3	31623	10^6	31622777	10^9
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}

Chapter 10 Searching and Selection

- 10.1 Search an unsorted array
- 10.2 Search a sorted array
- 10.3 Selection

10.1 Search an Unsorted Array

- A sequential search of a list compares the desired item with the entries in the list in a sequential order, until it locates the desired item or returns without success.

10.1 Search an Unsorted Array

- An iterative sequential search of an unsorted array.

```
public static boolean  contains(Object[] A, Object target){
    for(int i=0; i<A.length; i++){
        if(A[i].equals(target))
            return true;
    }
    return false;
}
```


10.1 Search an Unsorted Array

- A recursive sequential search of an unsorted array.

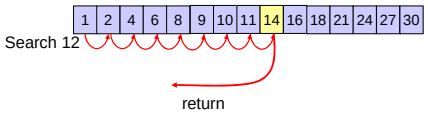
```
public static boolean contains(Object[] A, Object target){
    return search(A, target, 0, A.length);
}
private static boolean search(Object[] A, Object target, int start, int end){
    if(start > end)
        return false;
    if(A[start].equals(target))
        return true;
    return search(A, target, first+1, end);
}
```


10.1 Search an Unsorted Array

- Running time of a recursive sequential search of an array.
  - Best case O(1)
  - Worst case O(n)
  - Average case O(n)

10.2 Search a sorted array

- A sequential search of a sorted array
  - It can be more efficient if the data is sorted.



10.2 Search a sorted array

- A binary search of a sorted array

```
Binary search algorithm to search a sorted array for desiredItem
    mid = approximate midpoint in array A
    if(desiredItem == A[mid])
            return  true
    else if(desiredItem < A[mid])
        return the result of searching A[0] through A[mid-1]
    else if(desiredItem > A[mid])
        return the result of searching A[mid+1] through A[n-1]
```


10.2 Search a sorted array

```
public static boolean binarysearch(int[] A, int first, int last, int
    desiredItem){
    boolean found=false;
    if(first>last) return false;
    int mid = (first+last)/2;
    if(desiredItem==A[mid])
        found = true;
    else if(desiredItem<A[mid])
        found=binarysearch(A, first, mid-1, desiredItem);
    else if(desiredItem>A[mid])
        found=binarysearch(A, mid+1, last, desiredItem);
    return found;
}
```

10.2 Search a Sorted Array

- Running time of a recursive binary search of an array.
  - Best case O(1)
  - Worst case O(logn)
  - Average case O(logn)

10.3 Selection

- Selection problem
  - Given an array A and an integer k, find the kth smallest element.
- Any sorting algorithm can be used to solve the selection problem. However, since the selection problem does not require that the array be sorted, can we solve the problem without sorting the entire array?

10.3 Selection

- Recall the partition method used in quicksort:
  - Suppose we execute the partition procedure and the partition element — pivot — happens to be placed in the kth cell. The pivot element is the desired kth smallest element.
  - However, in general, the partition procedure does not always place the partition element in the kth cell.

10.3 Selection

- Observation: if the partition procedure places the partition element in the ith cell
  - If i<k, then the kth smallest element must be among the elements that are to the right of the partition element.
  - If i>k, then the kth smallest element must be among the elements that are to the left of the partition element.

10.3 Selection

- Implementing Random quick-select method
  - Random partition algorithm

    private static int randPartition(int[] A, int first, int last)

    This algorithm chooses an index m randomly, and then partitions the array A[first…last] by inserting val=A[m] at an index h where it would be if the array was sorted. Then return the index h.

10.3 Selection

```
private static int randPartition(int[] A, int first, int last ){
   int m = first+Math.random()*(last-first+1);
   swap(A, m, last);
   int leftIndex = first;
   int rightIndex = last-1;
   while (leftIndex<rightIndex){
     if(A[leftIndex]>A[last]) {
          swap(A, leftIndex, rightIndex)
          rightIndex--;
    } else leftIndex++;
   }
   int h = (A[leftIndex]<A[last])? (leftIndex+1): leftIndex;
   swap(A, h, last);
   return h;
}
```

10.3 Selection

- Select algorithm using random partition

  This algorithm searches an unsorted array A[first…last], finds the kth smallest element in the array and return its value.

```
public static int quickSelect (int[] A, int first, int last, int k){
    int h=randPartition(A, first, last);
    int val = 0;
   if(h==k) val=A[h];
   else if(h<k)
        val=quickSelect(A, h+1, last);
   else if(h>k)
        val=quickSelect(A, first, h-1);
   return val;
}
```


10.3 Selection

- Running time analysis of randomized quick-selection
  - Best case O(n)
  - Worst case O(n^2)
  - Average case O(n)

Chapter 11. Maps, Dictionaries and Sets

- 11.1 Maps
- 11.2 Hash tables
  - 11.2.1 Bucket arrays
  - 11.2.2 Hash fuctions
  - 11.2.3 Hash code
  - 11.2.4 Compression fuctions
- 11.2 Dictionaries
- 11.3 Sets

11.1 Maps

- A map stores key-value pairs(k, v), which we call entries, where k is the key and v is its corresponding value.
- Each key in a map must be unique.

11.1 Maps

- The Map ADT(based on java.util.Map)

int size()	Return the number of entries in the map
boolean isEmpty()	Test whether the map is empty
object get(Object k)	Return the value of an entry with key equal to k
object remove(Object k)	remove an entry with key equal to k and return its value
Object put(Object k, Object v)	If the map doesn't have an entry with key equal to k, then add entry(k, v); else, replace with v the existing value of the entry with key equal to k and return the old value
keySet()	Returns a set view of the keys contained in this map.
Values()	Returns a collection view of the values contained in this map.

11.2 Hash Tables

- The running time of map operations in an n-entry map is O(n).
- A hash table usually can perform these operations in O(1) expect time.

11.2.1 Bucket Array

- Is an array of size N, where each cell of A is thought of as a "bucket" which is a container of key-value pairs.
  - The keys are integers well distributed in the range of [0, N-1].
  - An entry with key k is inserted into the bucket A[k].

0	1	2	3	4	5	6	7	8	9
	(1, D)			(4, A) (4, N)			(7, F)		

11.2.1 Bucket Arrays

- Pros and cons
  - Pros: If the keys are unique integers, then each bucket holds at most one entry. Thus, the search, remove, insert operations run in O(1) time.
  - Cons
    - May be a waste of space. If N is much larger than the number of entries n.
    - The keys are required to be integers in the range[0, N-1], which is often not the case

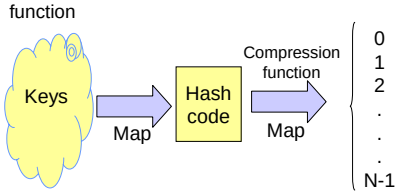
Solution – Use the bucket array in conjunction with a "good" mapping from the keys to the integers in the range [0,N-1].

11.2.2 Hash functions

- A hash function h maps each key k in the map to an integer in the range [0, N-1].
  We store the entry (k, v) in the bucket A[h(k)].
- Collision: If there are two or more keys with the same hash value, then two or more different entries will be mapped to the same bucket. A "good" hash function should minimize collision as much as possible.

Hash functions

- A hash function has two actions
  - Mapping the key to an integer, called the hash code
  - Mapping the hash code to an integer within the range of indices [0, N-1] of a bucket array – compression function



11.2.3 Hash Codes

- Hash codes in Java
  - Java Object class defines a default hashCode() method for mapping each Object instance to an 32-bit integer of type int
- Hash codes for common data types.
  - Casting to an integer
    - byte, short, int, char types
      Simply cast these types to int to get their hash codes
    - Float type
      - Float.floatToIntBits(x) //convert a float x to an integer.

11.2.3 Hash Codes

- Summing components
  - Long and double types that are 64-bit
    - 1. Simply cast a long integer down to a 32-bit integer by ignoring half of the information presented in the original value – may easy lead to Collisions.
    - 2. Sum an integer representation of the high-order bits with an integer representation of the low-order bits.

Summation hash code: We can view the binary representation of any object x as k-tuple(x_0, x_1, x_2, …, x_{k-1}) of integers, then the hash code for x can be formed as $\sum_{i=0}^{k-1} x_i$

11.2.3 Hash Codes

- Polynomial Hash Codes
  - The summation hash code is not a good choice for strings or other variable-length objects that can be viewed as tuples of the form

$$(x_0, x_1, x_2, ..., x_{k-1})$$

where the order of the elements are significant.

Example: "temp10" and "temp01" collide

"stop", "spot", "tops" and "pots" collide

11.2.3 Hash Codes

- Polynomial hash code:

Alternatively, we use as hash code the value

$$x_0 a^{k-1} + x_1 a^{k-2} + x_2 a^{k-3} + ... + x_{k-2} a + x_k$$

Where a is a nonzero constant, $a \neq 1$ And $x_0, x_1, x_2, ..., x_{k-1}$ are the components of an object.

By Horner's rule this polynomial can be written as

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + ... + a(x_2 + a(x_1 + ax_0))...))$$

11.2.3 Hash Codes

- Experimental results
  - Experimental studies suggest that 33, 37, 39, and 41 are particularly good choices for a when working with character strings that are English words. In a list of over 5000 English words, there are less than 7 cases of collision when taking these integers as a.

11.2.4 Compression Functions

- The division method

It maps an integer i to

|*i*| *mod N*.

where N is the size of the bucket array.

- The MAD method
  - MAD=Multiply add and divide

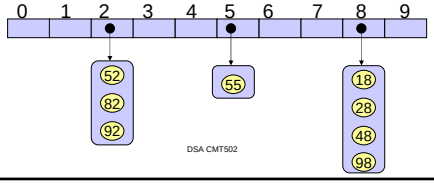
It maps an integer i to

|*ai+b*| *mod N*.

where N is a prime number and a>=0, b>=0 are integer constants randomly chosen at the time the compression function is determined so that a!=N

11.2.5 Collision-Handling Schemes

- Separate chaining
  - The bucket A[i] store a small map M_i , implemented using a list, holding entries (k, v) such that h(k)=i.
  - **Load factor**=n/N, where n is the number of entries in the map, N is the size of the bucket array.



11.2.5 Collision-Handling Schemes

- Open Addressing Schemes
  - It requires
    - the load factor be always at most 1
    - the entries must be stored directly in the cells of the bucket array itself
  - Linear Probing
    - If we try to insert an entry (k, v) into a bucket A[i] that is already occupied, we try next at A[(i+1)modN], and so on, until we find an empty bucket.
    - It saves space, but it causes clustering and slows down the search operation

11.2.5 Collision-Handling Schemes

- Quadratic Probing
  - If we try to insert an entry (k, v) into a bucket A[i] that is already occupied, we iteratively trying the bucket A[(i+f(j)) mod N), for j=0, 1, 2, 3, …, where $f(j)=j^2$, until we find an empty bucket.
  - It complicates the removal operation and also causes some amount of clustering.

11.3 Dictionaries

- A dictionary stores key-value pairs (k, v), which we call entries, where k is the key and v is its corresponding value.
- It allows for multiple entries to have the same key.
- There are two types of dictionaries
  - Unordered dictionary
  - Ordered dictionary

11.3 Dictionaries

- Dictionary ADT

int size()	Return the number of entries in D
boolean isEmpty()	Test whether D is empty
object find(Object k)	Return the value of an entry with key equal to k
Iterator findAll(Object k)	Return the iterator of all entries with key equal to k
Entry remove(Entry e)	remove from D the entry e, returning the removed entry.
Entry insert(Object k, Object v)	Insert an entry with key k and value e into D, returning the entry created.
Iterator entries()	Return an iterator of entries stored in D

11.4 Set

- A set is a container of distinct objects.
  - No duplicated elements
  - No explicit keys
  - No explicit order
- Union, intersection and subtraction of two sets A and B:
  - $$A \bigcup B = \{x : x \in A or x \in B)$$
    $$A \bigcap B = \{x : x \in A and x \in B)$$
    $$A - B = \{x : x \in A and x \notin B)$$

11.4 Set

- Set ADT
  - The fundamental methods
    - Union(A, B): return the union of A and B.
    - intersect(A, B): return the intersection of A and B.
    - subtract(A, B): return the difference of A and B.

11.4 Set

- A simple set implementation
  - Storing a set into an ordered sequence.
  - Union, intersection and subtraction operations can be implemented by using a generic merging algorithm that takes two sorted sequences and constructs a sequence representing the output set.

- A implementation of union operation.

```
Input: sequence A and sequence B
Output: sequence C
union(A, B){
  Initiate an empty set C
   indexA =0
  indexB=0
  while(indexA < length of A and indexB < length of B){
        a is the element at indexA in sequence A
        b is the element at indexB in sequence B
        if(a==b){ copy a to the end of C , indexA++, indexB++}
        else if (a>b){ copy b to the end of C, indexB++}
        else { copy a to the end of C, indexA++}
   }
   if(indexA < length of A)
        copy the rest of the elements in A to the end of C
   else if(indexB < length of B)
        copy the rest of the elements in B to the end of C
   return C
}
```


- A implementation of intersection operation.

```
Input: sequence A and sequence B
Output: sequence C

intersect (A, B){
      initiate an empty set C
      indexA =0
      indexB=0
      while(indexA < length of A and indexB < length of B){
            a is the element at indexA in sequence A
            b is the element at indexB in sequence B
            if(a==b){
                  copy a to the end of C , indexA++, indexB++
            }else if (a>b)   indexB++
            else    indexA++
       }
       return C
}
```


- A implementation of subtraction operation.

```
Input: sequence A and sequence B
Output: sequence C
subtract (A, B){
initiate an empty set C
indexA =0
      indexB=0
      while(indexA < length of A and indexB < length of B){
            a is the element at indexA in sequence A
            b is the element at indexB in sequence B
            if(a==b){ indexA++, indexB++}
            else if (a>b)   indexB++
            else  { copy a to the end of C,  indexA++}
       }
       if(indexA<length of A)
             copy the rest of the elments in A to the end of C.
       return C
}
```


11.4 Set

- Performance of generic merging
  - Assume n_a is size of A, n_b is the size of B, the total running time is $O(n_a+n_b)$.
  - The set which is implemented with an ordered sequence and a generic merging scheme supports union, intersection and subtraction in O(n) time.

Chapter 12 Graphs

- 12.1 Graph terminology
- 12.2 Graph representation
- 12.3 Graph traversals
  - 12.3.1 Depth-first search
  - 12.3.2 Breadth-first search
  - 12.3.3 Topological order
  - 12.3.4 Find a path

12.1 Graph Terminology

- A **graph** G is a set V of **vertices** and a collection E of pairs of vertices from V, called **edge**s.
- An edge *(u, v)* is said to be **directed** from *u* to *v* if the pair *(u, v)* is ordered, with *u* proceeding *v*.
- An edge *(u, v)* is said to be **undirected** if the pair *(u, v)* is unordered.

12.1 Graph Terminology

- A graph is an **undirected graph** if all its edges are undirected.
- A graph is an **directed graph** (or **digraph**) if all its edges are directed.
- A graph is an **mixed graph** if it has both undirected and directed edges.

12.1 Graph Terminology

- The two vertices joined by an edge are called the **end vertices** of the edge, and they are **adjacent**.
- An edge is said to be **incident** on a vertex if the vertex is one of the edge's end vertices.
- The **degree** of a vertex v, denoted **deg(v),** is the number of the incident edges of v.

12.1 Graph Terminology

- The **outgoing edges** of a vertex are the directed edges whose origin is that vertex.
- The **incoming edges** of a vertex are the directed edges whose destination is that vertex.
- The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted **indeg(v)** and **outdeg(v),** respectively.

12.1 Graph Terminology

- A **path** is a sequence of edges that connect two vertices in a graph. The **length of a path** is the number of edges it comprises.
- If the path does not pass through any vertex more than once, it is a **simple path**.
- A **cycle** is a path that begins and ends at the same vertex. A **simple cycle** passes through other vertices only once.
- A **acyclic graph** is a graph without any cycles.

12.1 Graph Terminology

- A **directed path** is a path such that all the edges are directed and are travelled along their direction.
- A **directed cycle** is a cycle such that all the edges are directed and are travelled along their direction.

12.1 Graph Terminology

- A **subgraph** is a portion of a graph that is itself a graph.
- A **spanning subgraph** of graph G is a subgraph of G that contains all the vertices of G.
- A **connected graph** is a graph, in which, for any two vertices, there is a path between them.
- A **maximal subgraph** is a subgraph with the maximum possible number of edges (every edge which is in the original and has both endpoints in the vertex set of the subgraph).
- If a graph is not connected, its maximal connected subgraphs are called the **connected components** of G.

12.1 Graph Terminology

- A **forest** is a graph without cycles.
- A **tree** is a connected forest. (The trees we have learned about earlier can be called **rooted trees**, and the trees in this chapter are called **free trees**).
- A **spanning tree** of a graph is a spanning subgraph that is a tree.

12.1 Graph Terminology

- A **weighted graph** is a graph that has a numeric label w(e) associated with each edge e, called the **weight** of edge e.

12.1 Graph Terminology

- Properties

  Let G be an undirected graph with n vertices and m edges

  If G is connected, then m>=n-1

  If G is a tree, then m=n-1

  If G is a forest, then m<=n-1

12.2 Graph ADT

vertices()	Return an iterator of all the vertices of a graph
edges()	Return an iterator of all the edges of a graph
incidentEdges(v)	Return an iterator of the edges incident upon vertex v.
opposite(v, e)	Return the end vertex of edge e from vertex v.
endVertices(e)	Return an array storing the end vertices of edge e.
areAdjacent(v, w)	Test whether vertices v and w are adjacent
replace(v, x)	Replace the element stored at v with x

12.2 Graph ADT

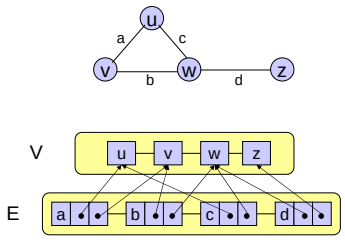
insertVertex(x)	Insert and return a new vertex storing element x
insertEdge(v, w, x)	Insert and return a new undirected edge with end vertices v and w and storing element x.
removeVertex(v)	Remove vertex v and all its incident edges and return the element stored in v
removeEdge(e)	Remove edge e and return the element stored at e.

12.2 Graph Representations

- The edge list structure
  - In a graph representation, there are two containers V and E. V stores all the vertex objects of the graph and E stores all the edge objects of the graph.
    - A vertex object stores an element
    - A edge object stores an element and two references to the vertex objects representing its end vertices.

12.2 Graph Representations

- Visualize the edge list structure



12.2 Graph Representations

- Performance of the edge list structure (m is the number of edges, n is the number of vertices)

Operation	Time
Vertices()	O(n)
Edges()	O(m)
endVertices(), opposite()	O(1)
incidentEdges(), areAdjacent()	O(m)
Replace()	O(1)
InsertVertex(), insertEdge(), removeEdge()	O(1)
removeVertex()	O(m)

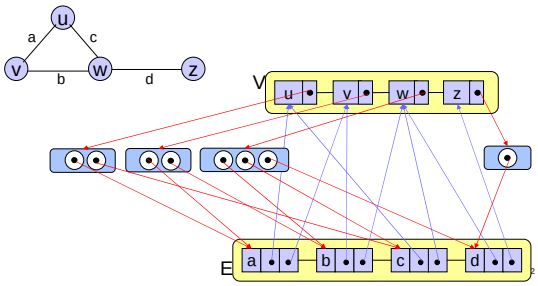
- Space usage is O(m+n)

12.2 Graph Representations

- The adjacency list structure
  - In a graph representation, there are two containers V and E. V stores all the vertex objects of the graph and E stores all the edge objects of the graph.
    - A vertex object stores an element and a reference to an adjacency list which stores references to the edges incident on the vertex
    - An edge object stores an element and two references to the vertex objects representing its end vertices.

12.2 Graph Representations

- Visualize the Adjacency list structure



12.2 Graph Representations

- Performance of the adjacency list structure (m is the number of edges, n is the number of vertices)

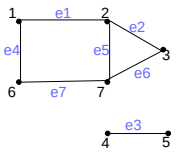
Operation	Time
Vertices()	O(n)
Edges()	O(m)
endVertices(), opposite()	O(1)
incidentEdges(v)	O(deg(v))
areAdjacent(v, w)	O(min(deg(v), deg(w))
Replace()	O(1)
InsertVertex(), insertEdge(), removeEdge()	O(1)
removeVertex()	O(deg(v))

- Space usage is O(m+n)

12.2 Graph Representations

- The adjacency matrix structure
  - This method uses an adjacency matrix to represent a graph
    - The vertices in the graph needs to be ordered so that each vertex is associated with an index number.
    - In the adjacency matrix representing the graph, the entry in row i, column j is 1 if (i, j) is an edge, or 0 if (i, j) is not an edge.

12.2 Graph Representations



Graph

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 6 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency matrix

12.2 Graph Representations

- Performance of the adjacency matrix structure (m is the number of edges, n is the number of vertices)

Operation	Time
Vertices()	O(n)
Edges()	O(m)
endVertices(), opposite()	O(1)
incidentEdges(v)	O(n+deg(v))
areAdjacent(v, w)	O(1)
Replace()	O(1)
insertEdge(), removeEdge()	O(1)
InsertVertex(), removeVertex()	$O(n^2)$

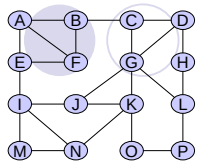
- Space usage is $O(m+n^2)$

12.3 Graph Traversals

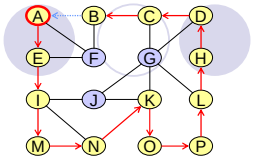
- A traversal is a systematic procedure for exploring a connected graph by examining all its vertices and/or edges.
  - It is efficient if it visits all the vertices and edges in linear time (which is proportional to the number of all the vertices and/or edges).

12.3.1 Depth-First Search

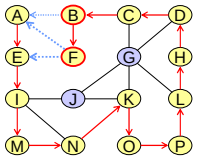
- A depth-first search visits a vertex, then a neighbour of the vertex, a neighbour of the neighbour, and so on, advancing as far as possible from the original vertex. It then backs up by one vertex and consider another neighbour.
- Example



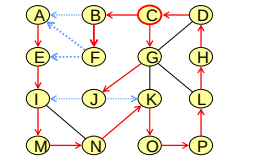
(a) Stack S:
Queue Q:



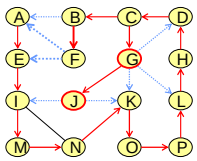
(b) Stack S: BCDHLPOKNMIEA
Queue Q: AEIMNKOPLHDCB



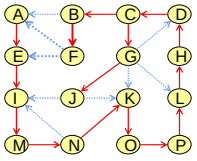
(c) Stack S: FBCDHLPOKNMIEA
Queue Q: AEIMNKOPLHDCBF



(d) Stack S: JGCDHLPOKNMIEA
Queue Q: AEIMNKOPLHDCBFGJ



(e) Stack S: CDHLPOKNMIEA
Queue Q: AEIMNKOPLHDCBFGJ



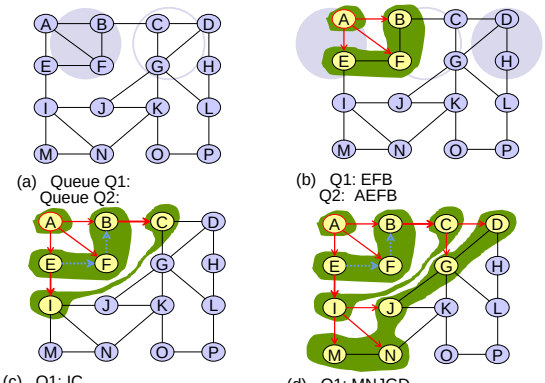
(f) Stack S:
Queue Q: AEIMNKOPLHDCBFGJ

DFS Algorithm

```
Input: A graph G, vertex v
Output: a new queue Q for the resulting traversal order
DFSTraversal(G,v){
  initiate a stack S and a queue Q
  mark v as visited
  S.push(v)
  Q.enqueue(v)
  topVertex = v;
  while(!S.isEmpty()){
    if(topVertex has an unvisited neighbour){
      nextNeighbour = next unvisited neighbour of topVertex
      mark nextNeighbour as visited
      Q.enqueue(nextNeighbour)
      S.push(nextNeighbour)
    } else
      topVertex = S.pop()
  }
  return Q
}
```


12.3.2 Breadth-First Search

- Given an original vertex, a breadth-first traversal visits the origin and the origin 's neighbours. It then considers each of these neighbours and visit their neighbours.
- Example

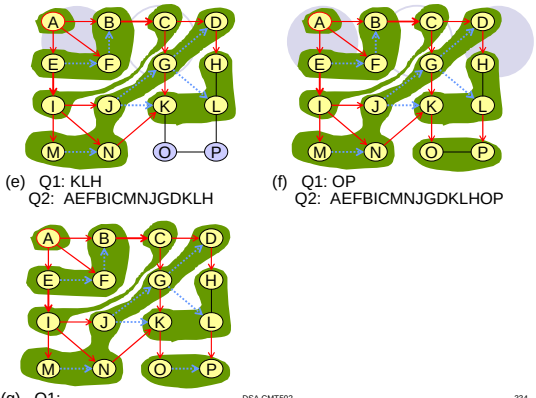


(a) Queue Q1:
Queue Q2:

(b) Q1: EFB
Q2: AEFB

(c) Q1: IC
Q2: AEFBIC

(d) Q1: MNJGD
Q2: AEFBICMNJGD



(e) Q1: KLH
Q2: AEFBICMNJGDKLH

(f) Q1: OP
Q2: AEFBICMNJGDKLHOP

(g) Q1:
Q2: AEFBICMNJGDKLHOP

BFS Algorithm

```
Input: A graph G, vertex v
Output: a new queue Q for the resulting traversal order
BFSTraversal(G,v){
  initiate two queues Q1 and Q2
  mark v as visited
  Q1.enqueue(v)
  Q2.enqueue(v)
  while(!Q1.isEmpty()){
    topVertex = Q1.dequeue();
    while(topVertex has an unvisited neighbour){
      nextNeighbour = next unvisited neighbour of topVertex
      Q1.enqueue(nextNeighbour)
      mark nextNeighbour as visited
      Q2.enqueue(nextNeighbour)
    }
  }
  return Q2
}
```


12.3.3 Topological Order

- In a directed graph without cycles, we can arrange the vertices so that vertex a precedes vertex b whenever a directed edge exists from a to b. The order of the vertices in this arrangement is called a **topological order**.

12.3.3 Topological Order

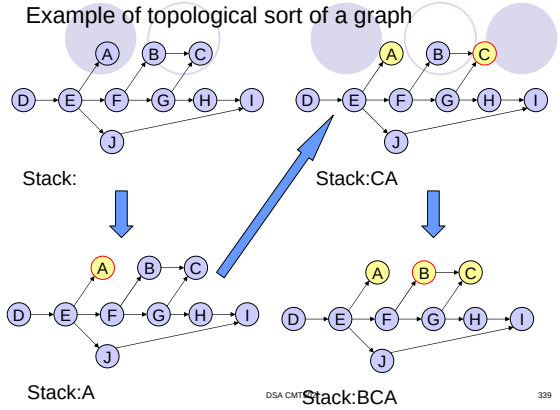
- In a directed graph without cycles, we can arrange the vertices so that vertex a precedes vertex b whenever a directed edge exists from a to b. The order of the vertices in this arrangement is called a **topological order**.
- **Topological sort**—discovers the topological order of a graph.

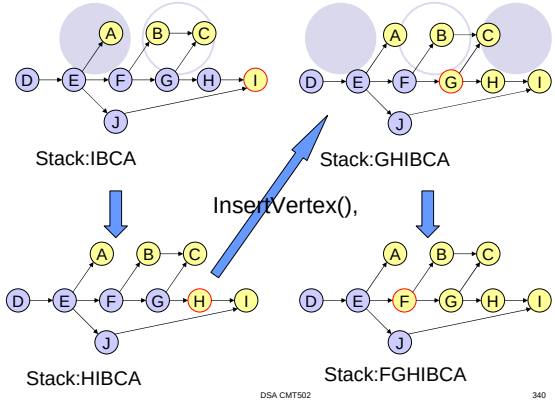
12.3.3 Topological Order

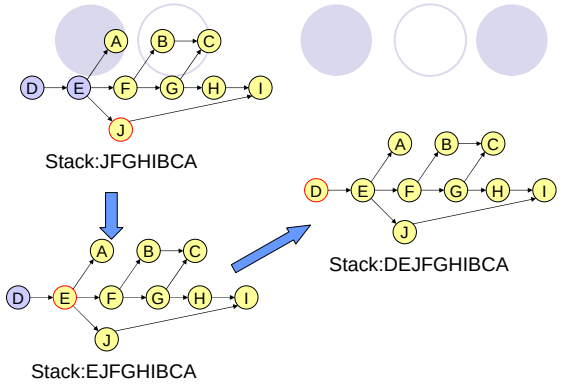
Topological sort algorithm

```
Input: acyclic directed graph G
output: a stack S for the resulting traversal order
 getTopologicalOrder(G){
    n=number of vertices in G
    initiate a new stack S
    nextVertex=a vertex that has no successor.
   for(i=0; i<n; i++){
        mark nextVertex visited
        S.push(nextVertex)
        nextVertex=a vertex that either has no successor or all his
                successors have been visited
    }
    return S
}
```


Example of topological sort of a graph

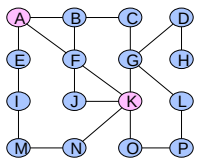






12.3.4 Finding a Path

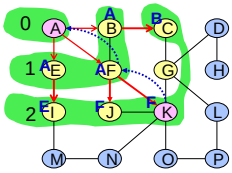
- 12.3.4.1 The shortest path in an unweighted graph



A→E→I→M→N→K
A→F→J→K
A→F→K
A→B→F→K
A→B→C→G→K

12.3.4.1 The shortest path in an unweighted graph

- Enhance breadth-first traversal to solve the problem



algorithm for getting shortest path in an unweighted graph

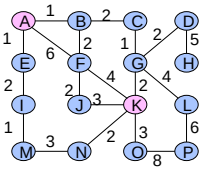
```
Input:  graph G, vertices originVertex, endVertex
Output: a stack S for the resulting traversal order
 getShortestPath(G, originVertex, endVertex){
     done = false
     initiate a queue Q to hold neighbours
     mark originVertex as visited
     Q.enqueue(oringinVertex)
     while(!done && !Q.isEmpty()){
         frontVertex=Q.dequeue();
         while(!done && frontVertex has unvisited neighbours){
             nextNeighbour = next unvisited neighbour of frontVertex
             mark nextNeighbour as visited
             set the predecessor of nextNeighbour to frontVertex
              Q.enqueue(nextNeighbour)
              if(nextNeighbour==endVertex) done=true
         }
     }
```



```
     S = a new stack of vertices
     S.push(endVertex)
     while(endVertex has a prodecessor){
         endVertex = prodecessor of endVertex
          S.push(endVertex)
     }
     return S
}
```


12.3.4.2 The shortest path in a weighted graph

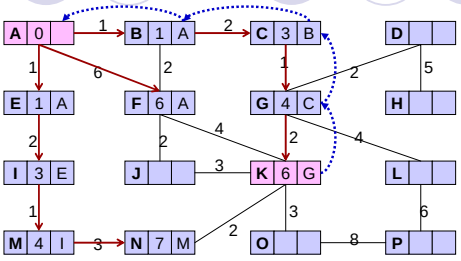
- The shortest path in a weighted graph is not necessarily the one with the fewest edges, but the one with the smallest edge-weight sum.



path	weight
A→E→I→M→N→K	8
A→F→J→K	11
A→F→K	10
A→B→F→K	7
A→B→C→G→K	6

12.3.4.2 The shortest path in a weighted graph

- Developing the algorithm:
  - based on a breadth-first traversal
  - uses a priority queue.
    - Each entry in the priority queue is an object that contains
      - A vertex
      - The cost of the path to that vertex from the origin vertex
      - The previous vertex on that path
    - The queue uses the cost of the path as priority. The less cost has higher priority.



algorithm for getting shortest path in a weighted graph

```
Input:  graph G, vertices originVertex, endVertex
output: a stack S for the resulting traversal order
getShortestPath(G, originVertex, endVertex){
    initiate a priority queue Q
    set originVertex as visited
    Q.enqueue(new pathEntry(originVertex, 0, null))
    while(Q.isEmpty()){
        frontEntry = Q.dequeue()
        frontVertex = vertex in frontEntry
        if(frontVertex equals endVertex)  break
        while(frontVertex has an unvisited neighbour){
            nextNeighbour = next unvisited neighbour of frontVertex
            set the cost of path to nextNeighbour as
                (weight of edge between frontVertex and
                nextNeighbour + cost of path to frontVertex)
            set the processor of nextNeighbour as frontVertex
            set nextNeighbour as visited
            add nextNeighbour to Q
        }
    }
```



```
    S = a new stack of vertices
    S.push(endVertex)
    while(endVertex has a prodecessor){
        endVertex = prodecessor of endVertex
        S.push(endVertex)
    }
    return S
}
```


Chapter 13 Greedy Algorithms

- 13.1 Coin changing
- 13.2 Kruskal algorithm
- 13.3 Prim's algorithm
- 13.4 Dijkstra's algorithm

Chapter 13 Greedy Algorithms

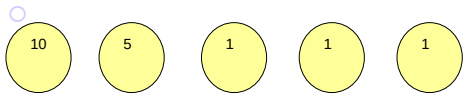
- A **greedy algorithm** builds a solution to a problem in steps. In each step, it adds a part of the solution which is the best available based on a **greedy rule**.

13.1 Coin Changing Problem

- Suppose we want to make change for an amount A using the fewest number of coins. Suppose further that the available denominations are 1, 5 and 10.
- One of the greedy rules: select the largest denomination available.

13.1 Coin Changing Problem

- Example: A=18.



13.1 Coin Changing Problem

- Greedy coin changing algorithm: this algorithm makes change for an amount A using coins of denominations
  denom[1]>denom[2]>…>denom[n]=1

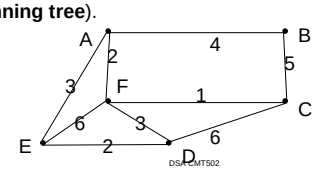
```
Input:  amount A, denom
Output: A queue storing the selections in order
Greedy-coin-changing(A, denom){
    i=1;
    Initiate a queue Q
    while(A>0){
        c=A/denom[i]
        if(c>0){
          for(j=0; j<c; j++) Q.enqueue(denom[i])
          A = A-c*denom[i]
          i++;
        }
    }
    return Q
}
```


13.1 Coin Changing Problem

- Running time analysis
  - The running time for the greedy coin changing algorithm is O(n), where n is the number of the denominations of coins.

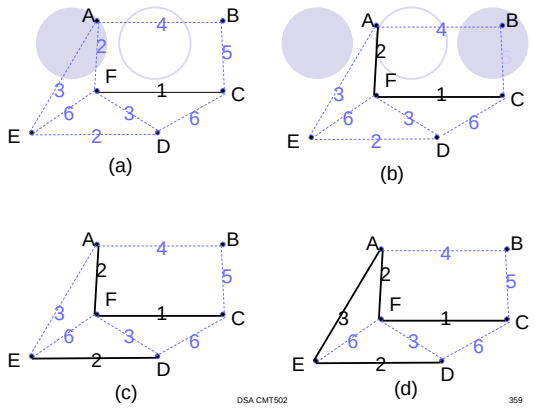
13.2 Kruskal Algorithm

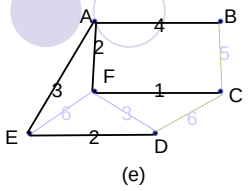
- A problem: The following graph shows six cities A, B, C, D, E and F and the costs (in hundreds of thousands of pounds) of rebuilding roads between them. We need to find out the cheapest way to rebuild enough roads so that each pair of cities will be connected. That is, to find a spanning tree with minimum weight (**minimal spanning tree**).



13.2 Kruskal Algorithm

- Kruskal algorithm is a greedy algorithm for finding a minimal spanning tree in a connected weighted graph G.
  - It begins with all the vertices of G and no edges.
  - It then applies the greedy rule: Add an edge of minimum weight that does not make a cycle.





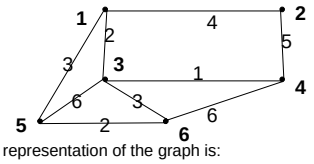
The cost of this spanning tree is 12.

13.2 Kruskal Algorithm

- Implementing the Kruskal Algorithm
  - Represents graph as a list of edges and their weights.
  - Sorts the edges in non-decreasing order by weight and examines them in sorted order.
  - Determines whether adding an edge would create a cycle.

13.2 Kruskal Algorithm

- Example: Find the minimal spanning tree in the following graph.



The representation of the graph is:
(1,2,4)(1,3,2)(1,5,3)(2,4,5)(3,4,1)(3,5,6)(3,6,3)(4,6,6)(5,6,2)
Where (a, b, w) is interpreted as edge (a, b) of weight w.

13.2 Kruskal Algorithm

First of all, we sort the edges in non-descendent order by weight
(3,4,1)(1,3,2)(5,6,2)(1,5,3)(3,6,3)(1,2,4)(2,4,5)(3,5,6)(4,6,6)

When the Kruskal algorithm starts, no edges have been selected, so we put all the vertices into different sets
{1} {2} {3} {4} {5} {6}

The first edge (3,4) is selected, and the sets to which vertices 3 and 4 belong are merged:
{1} {2} {3,4} {5} {6}

Next edge (1,3) is selected, and the sets to which vertices 1 and 3 belong are merged:
{1,3,4} {2} {5} {6}

13.2 Kruskal Algorithm

Next edge (5,6) is selected, and the sets to which vertices 5 and 6 belong are merged:
{1,3,4} {2} {5,6}

Next edge (1,5) is selected, and the sets to which vertices 1 and 5 belong are merged:
{1,3,4, 5,6} {2}

Next edge (3,6) is examined and rejected, because its vertices 3 and 6 belongs to the same set.

Finally edge (1,2) is selected, and the sets to which vertices 1 and 2 belong are merged:
{1,3,4, 5,6, 2}

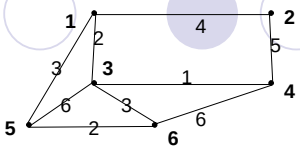
The algorithm

```
Input: edgelist, n
//edgelist is an array of edges. The members of edge are its end vertices
  and its weight. N is the number of vertices.
Output:
Kruskal(edgelist, n){
   sort edgelist in a non-descendent order
   make a set for each vertex
   i=0
   count =n
   while( count>1){
      (a, b) = the edge represented in edgelist[i]
      if (vertices a and b are in different sets){
            println ("("+a+", "+b+")")
            merge the two sets
            count - -
      }
      i++
   }
}
```

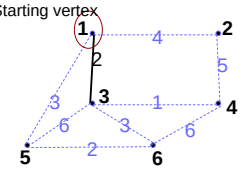

13.3 Prim's Algorithm

- Prim's algorithm is another greedy algorithm for finding a minimal spanning tree in a connected weighted graph G.
  - It begins with a start vertex and no edges.
  - It then applies the greedy rule: Add an edge of minimum weight that has one vertex in the current tree and the other not in the current tree.

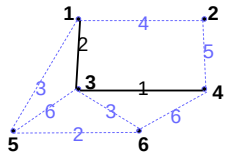
Find the minimum spanning tree



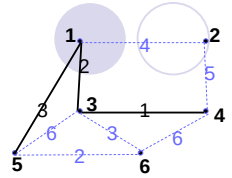
Starting vertex



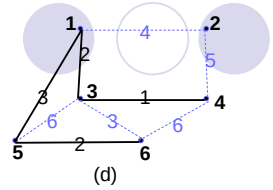
(a)



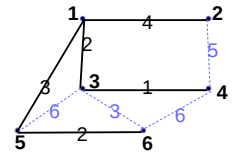
(b)



(c)



(d)



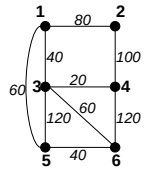
(e)

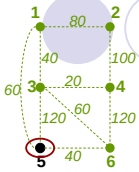
13.4 Dijkstra's Algorithm

- Dijkstra's algorithm is a greedy algorithm that finds the shortest paths from a designed vertex s to all other vertices in non-decreasing order of length.
  - The first path found is from s to s of length 0.
  - It then applies the greedy rule: Among all of the vertices that can extend a shortest path already found by one edge, choose the one that results in the shortest path.

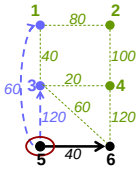
13.4 Dijkstra's Algorithm

- Example: finds the shortest paths from vertex 5 to all other vertices in non-decreasing order.

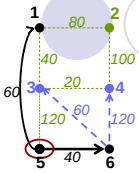




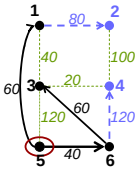
Shortest path	Path length
5	0



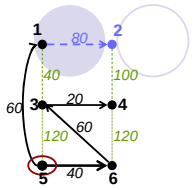
Shortest path	Path length
5	0
5→6	40



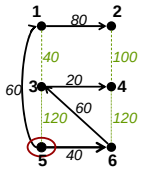
Shortest path	Path length
5	0
5→6	40
5→1	60



Shortest path	Path length
5	0
5→6	40
5→1	60
5→6→3	100



Shortest path	Path length
5	0
5→6	40
5→1	60
5→6→3	100
5→6→3→4	120



Shortest path	Path length
5	0
5→6	40
5→1	60
5→6→3	100
5→6→3→4	120
5→1→2	140