Q&A

- When do you write test cases for “black box” testing? Before or after implementation?
- Explain why array implementations of queue are slow.
- Linked list allows data removal by?
- Define references/pointers.
- What is the purpose of garbage collection in memory management?

Summary

- Testing collection implementations
- Queues
- Motivation for linked lists
- Linked structures for implementing Collections

Readings

- [Mar07] Read 3.5
- [Mar13] Read 3.5

Removing:


```
/** Remove the value from the list
    Assumes list is not empty, and value not in first node */
public void remove (E item, LinkedNode<E>list){
    if (list.next==null) return;        // we are at the end of the list
    if (list.next.value.equals(item) )
        list.next =  list.next.next;
    else
        remove(item, list.next);
}
```

or

```
public void remove (E item, LinkedNode<E>list){
    LinkedNode<E> rest=list;
    while (rest.next != null  && !rest.next.value.equals(item))
        rest=rest.next;
    if (rest.next != null)
        rest.next = rest.next.next;
}
```

Why have a ‘rest’ to hold ‘list’?

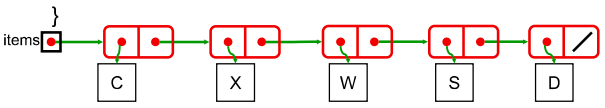
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){  /* recursive version */



}
```

or

```
public E lastValue (LinkedNode<E> list){/* iterative version */

}
```



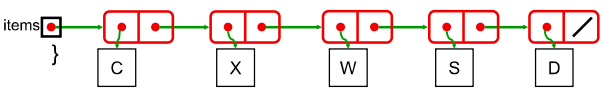
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){



}
```

or

```
public E lastValue (LinkedNode<E> list){
```




```
}
```

List using linked nodes with header

- LinkedList extends AbstractList
- Has fields for linked list of Nodes and count
- Has an inner class: Node, with public fields
- get(index), set(index, item),
  - loop to index'th node, then get or set value
- add(index, item), remove(index):
  - deal with special case of index == 0
  - loop along list to node one before index'th node (Why?), then add or remove
  - check if go past end of list
- remove(item),
  - deal with special case of item in first node (i.e. conversion into empty set after removal)
  - loop along list to node one before node containing item (Why?), then remove
  - check if go past end of list

More Linked Structures

Lectures 13-14

A Linked List class:

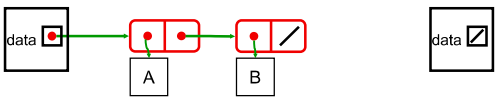

```
public class LinkedList <E> extends AbstractList <E> {
    private Node<E> data;
    private int count;
    public LinkedList(){
        data = null;
        count = 0;
    }
    /** Inner class:  Node */
    private class Node <E> {
        public E value;
        public Node<E> next;
        public Node(E val, Node<E> node){
            value = val;
            next = node;
        }
    }
```

Menu

- Linked structures for implementing Collections
- A collection class – Linked List
- Linked List methods

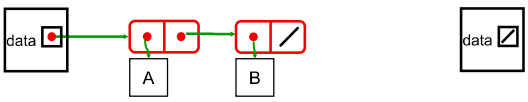
Linked List: get


```
public E get(int index){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index)  node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    return node.value;
}
```



How do you make a good list class

- Must have an object that represent the empty list as an object
  - separate "header" object to represent a list



Linked Collections: Cost

- Linked structures allow fast insertion and deletion
  Does it help?

Linked List:

- Cost of get / set:
- Cost of insert:
- Cost of remove:

Linked Set (items in sorted order):

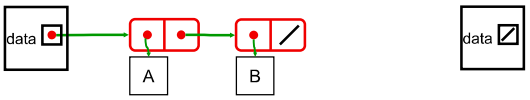
- Cost of contains:
- Cost of insert:
- Cost of remove

No advantage to Linked List?

Linked List: set


```
public E set(int index, E value){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index) node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    E ans = node.value;
    node.value = value;
    return ans;
}
```

Same as get

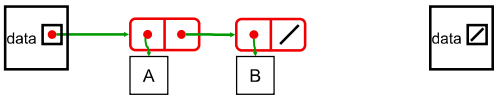


Menu

- Linked structures for implementing Collections
- A collection class – Linked List
- Linked List methods

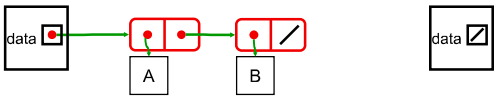
Linked List: add


```
public void add(int index, E item){
    if (item == null) throw new IllegalArgumentException();
    if (index==0){              // add at the front.
        data = new Node(item, data);
        count++;
        return;
    }
    Node<E> node=data;
    int i = 1;                  // position of next node
    while (node!=null && i++ < index) node=node.next;
    if (node == null) throw new IndexOutOfBoundsException();
    node.next = new Node(item, node.next);
    count++;
    return;
}
```



Linked List: remove


```
public boolean remove (Object item){
    if (item==null || data==null) return false;
    if (item.equals(data.value))   // remove the front item.
        data = data.next;
    else {          // find the node just before a node containing the item
        Node<E> node = data;
        while (node.next!=null && !node.next.value.equals(item))
            node=node.next;
        if (node.next==null)   return false;   // off the end
        node.next = node.next.next;   // splice the node out of the list
    }
    count--;
    return true;
}
```

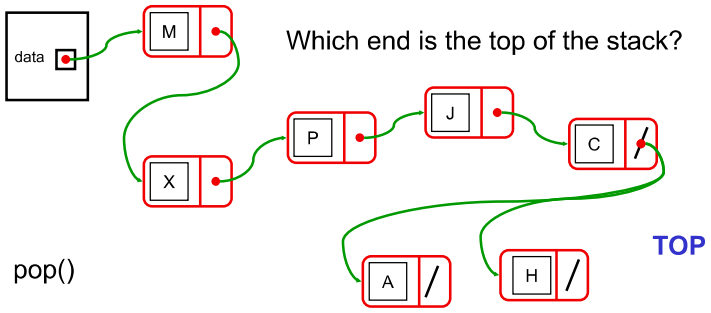


Linked Stacks and Queues

Lecture 15

A Linked Stack

- Implement a Stack using a linked list.



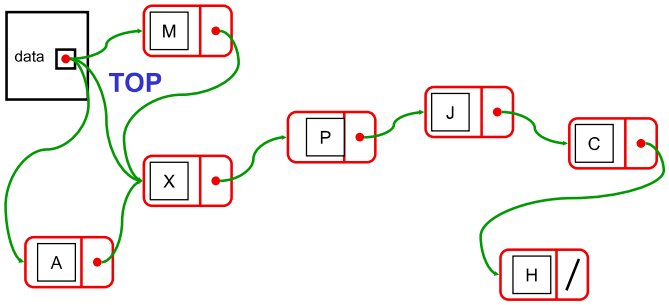
Which end is the top of the stack?

- pop()
- push("A")

Why is this a bad idea?

A Linked Stack

- Make the top of the Stack be the front of the list.



- pop()
- push("A")

Menu

- A Stack using a Linked List with a header
- A Queue using a Linked List with a header

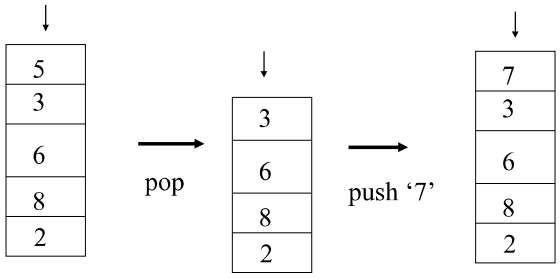
Implementing LinkedStack

- Use the LinkedNode class:

```
public class LinkedStack <E> extends AbstractCollection <E> {
    private Node<E> data = null;
    public LinkedStack(){ }
    public int size(){…
    public boolean isEmpty(){…
    public E peek(){…
    public E pop(){…
    public void push(E item){…
    public Iterator <E> iterator(){
```


Stacks (LIFO)

- insertions (push) & deletions (pop) only at the end (top)

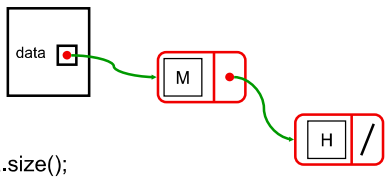


Application of Queues

- user job queue
- print spooling queue
- I/O event queue
- incoming packet queue
- outgoing packet queue

LinkedStack


```
public boolean isEmpty(){
    return data==null;
}
public int size () {
    if (data == null)   return 0;
    else                return data.size();
}
```

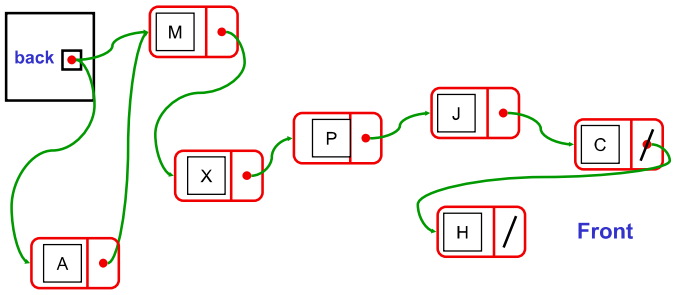


- Need size() method in Node class:

```
public int size (){
    int ans = 0;
    for (Node<E> rest = data; rest!=null; rest=rest.next)
        ans++;
    return ans;
}
```

A Linked Queue #1

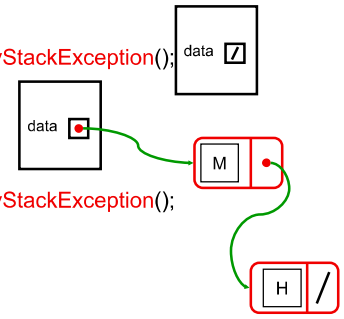
- Put the front of the queue at the end of the list



- offer("A")
- poll()

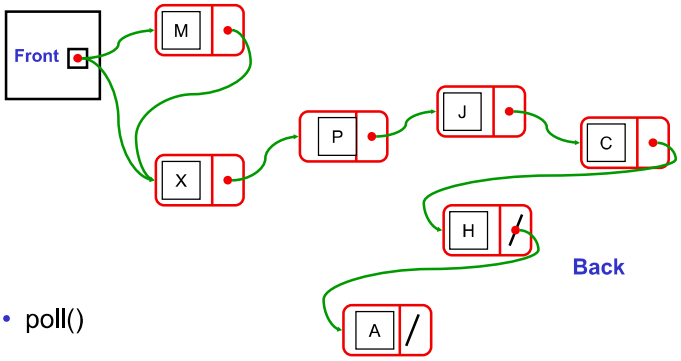
LinkedStack


```
public E peek(){
  if (data==null) throw new EmptyStackException();
  return data.value;
}

public E pop(){
  if (data==null) throw new EmptyStackException();
  E ans = data.value;
  data = data.next;
  return ans;
}
public void push(E item){
  if (item == null) throw new IllegalArgumentException();
  data = new Node(item, data);
}
public Iterator <E> iterator(){
    return new NodeIterator(data);
}
```



A Linked Queue #2

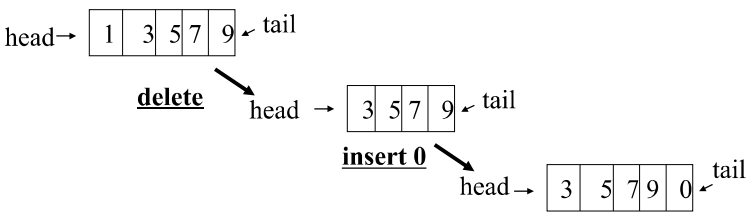
- Put the front of the Queue at the head of the list.



- poll()
- offer("A")

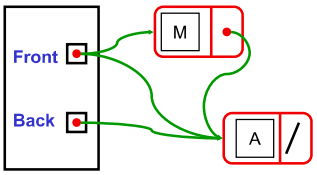
Queues (FIFO)

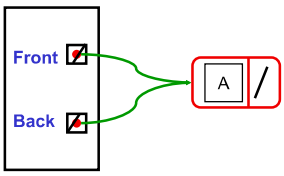
- Example: waiting lines
- Insertion at the end (tail), deletion from the front (head)



LinkedQueue

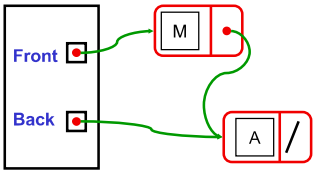
```
public E peek(){
   if (front==null)   return null;
   else               return front.value;
}

public E poll(){
   if (front==null) return null;
   E ans = front.value;
   front = front.next;
   if (front==null)  back = null;
   return ans;
}
```

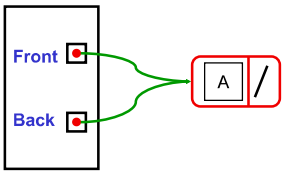




Exercise: work out the method body of 'offer'

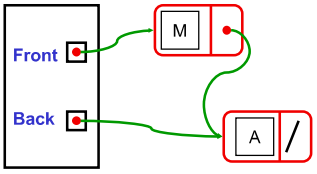
```
public boolean offer(E item){

}
```

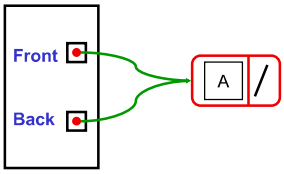




LinkedQueue

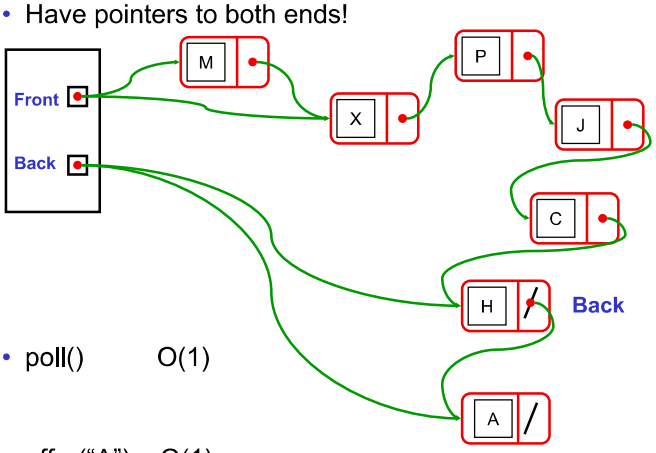
```
public boolean offer(E item){
   if (item == null) return false;
   if (front == null){
        back = new Node(item, null);
        front = back;
   }
   else {
        back.next = (new Node(item, null));
        back= back.next;
   }
   return true;
}
```





A Better Linked Queue

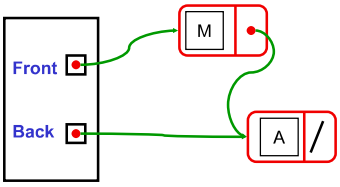
- Have pointers to both ends!



- poll() O(1)
- offer("A") O(1)

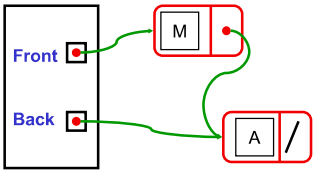
Implementing LinkedQueue

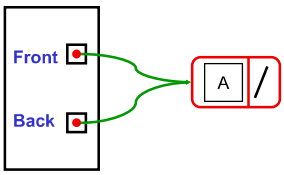
```
public class LinkedQueue <E> implements AbstractQueue <E> {

   private Node<E> front = null;
   private Node<E> back = null;

   public LinkedQueue(){ }

   public int size(){…

   public boolean isEmpty(){…

   public E peek(){…

   public E poll(){…

   public void offer(E item){…

   public Iterator <E> iterator(){
```

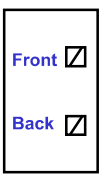


LinkedQueue

```
public boolean isEmpty(){
    return front==null;
}
public int size () {
   if (front == null)   return 0;
   else                 return front.size();
}
```







- Always three cases: 0 items, 1 item, >1 item

Linked Stack and Queue

- Uses a “header node”
  - contains link to head node, and maybe last node of linked list
- Important to choose the right end.
  - easy to add or remove from head of a linked list
  - hard to add or remove from the last node of a linked list
  - easy to add to last node of linked list if have pointer to tail
- Linked Stack and Queue:
  - all main operations are O(1)
- Can combine Stack and Queue
  - addFirst, addLast, removeFirst
  - also need removeLast to make a “Deque” (double-ended queue)
    ⇒ need doubly linked list (why?)
  - See the java “LinkedList” class.

Summary

- A Stack using a Linked List with a header
- A Queue using a Linked List with a header

Readings

- [Mar07] Read 3.6, 3.7, 6.2
- [Mar13] Read 3.6, 3.7, 6.2

ArraySet and Binary Search

Lecture 16

Menu

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

ArrayList costs: Summary

• get	O(1)	
• set	O(1)	
• remove	O(n)	
• add (at i)	O(n)	(worst and average)
(have to shift up may have to double capacity)		
• add (at end)	O(1)	(most of the time)
(when doubles cap)	O(n)	(worst)
	O(1)	(amortised average) (if doubled each time)

ArraySet costs

What about ArraySet:

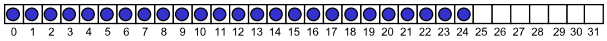
- Order is not significant

  ⇒ add() can choose to put a new item anywhere. where?

  ⇒ can reorder when removing an item. how?

- **Duplicates not allowed.**

  ⇒ must check if item already present before adding



ArraySet algorithms

Contains(value):

```
search through array,
    if value equals item
        return true
return false
```

Costs?

Add(value):

```
if  not contains(value),
    place value at end, (doubling array if necessary)
    increment size
```

Remove(value):

```
search through array
    if value equals item
        replace item by item at end. (why?)
        decrement size
        return
```

ArraySet costs

Costs:

- contains, add, remove: O(n)
- Question:
- How can we speed up the search?

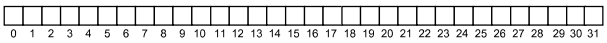
Binary Search

```
private boolean contains(Object item){
    Comparable<E> value = (Comparable<E>) item;
    int low = 0;                          // min possible index of item
    int high = count-1;                   // max possible index of item
                                          // item in [low .. high]  (if present)
    while (low <= high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data[mid]);
        if (comp == 0)                    // item is present
            return true;
        if (comp < 0)                     // item in [low .. mid-1]
            high = mid - 1;               // item in [low .. high]
        else                              // item in [mid+1 .. high]
            low = mid + 1;                // item in [low .. high]
    }
    return false;    // item in [low .. high] and low > high,
                     // therefore item not present
}
```

low mid high

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?



- Iteration Size of range Cost of iteration

  1 n

  2

  k 1

Time complexity

Let T(n) denote the time complexity of binary search algorithm on n numbers.

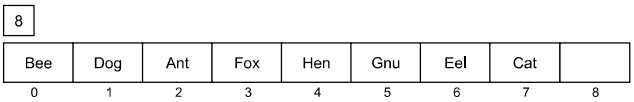
$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

We call this formula a **recurrence**.

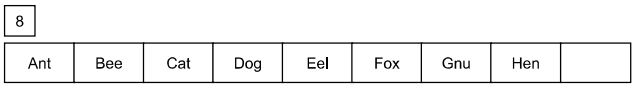
Making ArraySet faster.

All the cost is in the searching:

- Searching for “Eel”



- but if sorted…



Making ArraySet faster

- Binary Search: Finding “Eel”

- If the items are sorted (“ordered”), then we can search fast
  - Look in the middle:
    - if item is middle item ⇒ return
    - if item is before middle item ⇒ look in left half
    - if item is after middle item ⇒ look in right half

Divide and Conquer

One of the **best-known** algorithm design techniques.

Idea:

- A problem instance is ***divided*** into several **smaller** instances of the same problem, ideally of about same size
- The smaller instances are **solved**, typically **recursively**
- The solutions for the smaller instances are ***combined*** to get a solution to the original problem

Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by MI.

Assume true for all $n' < n$ [assume $T(n/2) \le 2 \log (n/2)$]

$T(n) = T(n/2) + 1$

$\le 2 \log (n/2) + 1$ ← *by hypothesis*

$= 2(\log n - 1) + 1$ ← ***log(n/2) = log n - log 2***

$< 2 \log n$

i.e., $T(n) \le 2 \log n$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \le 2 n \log n$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \le 2 n \log n$

Assume true for all $n' < n$ [assume $T(n/2) \le 2 (n/2) \log(n/2)$]

$T(n)$

$\le 2 (2 (n/2) \log (n/2)) + n$

$= 2 n (\log n - 1) + n$

$= 2 n \log n - 2n + n$

$\le 2 n \log n$

For the base case when n=2,
L.H.S = T(2) = 2T(1)+2 = 4,
R.H.S = 2 * 2 log 2 = 4
L.H.S ≤ R.H.S

i.e., $T(n) \le 2 n \log n$

Recurrence

A recurrence is an equation or inequality that describes a function in terms of ***its value on smaller inputs***.

E.g., $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n > 1 \end{cases}$

To **solve** a recurrence is to derive ***asymptotic bounds*** on the solution

$Log_2(n)$ or $log(n)$

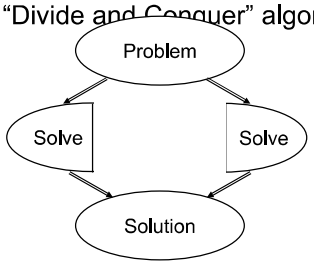
The number of times you can divide a set of n things in half.

log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double n, you add one step to the cost! (why?)

- log(2n) = log(n)+log2 = log(n) +1
- Arises all over the place in analysing algorithms

Especially "Divide and Conquer" algorithms:



Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by MI.

Base case? When n=1, statement is FALSE!

L.H.S = **T(1) = 1** R.H.S = **c log 1 = 0** < L.H.S

Yet, when n=2,

L.H.S = T(2) = T(1)+1 = 2

R.H.S = 2 log 2 = 2

L.H.S ≤ R.H.S

Summary

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

Readings

- [Mar07] Read 4.3
- [Mar13] Read 4.3

ArraySet with Binary Search

ArraySet: unordered

- All cost in the searching: O(n)
  - contains: O(*n*)
  - add: O(*n*)
  - remove: O(*n*)

SortedArraySet: with Binary Search

- Binary Search is fast: O(log(*n*))
  - contains: O(log(*n*))
  - add:
  - remove:

- All the cost is in keeping it sorted!!!!

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order

- If you only have to construct the set once, and then many calls to contains(),
  then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))

- But, how do you construct the set fast?
  - Adding each item, one at a time

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> col){
    // Make space
    count=col.size();
    data = (E[]) new Object[count];
    // Put items from collection into the data array.
    col.toArray(data);
    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Sorting

Lecture 17

$Log_2(n)$ or $log(n)$:

The number of times you can divide a set of n things in half.

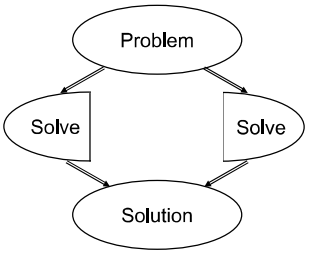
log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double n, you add one step to the cost!

$$2^{\log(x)} = x$$

- Arises all over the place in analysing algorithms

  Especially "Divide and Conquer" algorithms:



Menu

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

SortedArraySet algorithms

Contains(value):
index = findIndex(value),
return (data[index] equals value)

Add(value):
index = findIndex(value),
if index == count or data[index] not equal to value,
double array if necessary
move items up, from position count-1 down to index
insert value at index
increment count

Remove(value):
index = findIndex(value),
if index < count and data[index] equal to value
move items down, from position index+1 to count-1
decrement count

Assume data is sorted

Binary Search: Cost

- What is the cost of searching if n items in set?
  - key step = ?

Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

Sets with Binary Search

ArraySet: unordered

- contains: O(n)
- add: O(n)
- remove: O(n)

⇒ All the cost is in the searching: O(n)

SortedArraySet: with Binary Search

- Binary Search is fast: O(log(n))
  - contains: O(log(n))
  - add:
  - remove:

⇒ All the cost is in keeping it sorted!!!!

Sort them!

Rat	Pig	Owl	Kea	Fox	Hen	Yak	Ant_1	Tui	Dog	Cat	Man	Jay	Bee	Eel	Gnu	Ant_2	Sow

73	3	6931	427	5	45	463	941	7273	64	9731	61	873	44	74	465	6929	75

How to do it?

Ways of sorting

- Selecting sorts:
  - Find the next largest/smallest item and put in place
  - Builds the correct list in order
- Inserting Sorts:
  - For each item, insert it into an ordered sublist
  - Builds a sorted list, but keeps changing it
- Compare and Swap Sorts:
  - Find two items that are out of order, and swap them
  - Keeps "improving" the list
- Radix Sorts
  - Look at the item and work out where it should go.
  - Only works on some kinds of values.
- …

Analysing Sorting Algorithms

- Efficiency
  - What is the (worst-case) order of the algorithm?
  - Is the average case much faster than worst-case?
- Requirements on Data
  - Does the algorithm need random-access data? (*vs* streaming data)
  - Does it need anything more than "compare" and "swap"?
- Space Usage
  - Can the algorithm sort in-place? or does it need extra space?

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order
- If you construct the set once but many calls to contains(), then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))
- But, how do you construct the set fast?

Why Sort?

- Constructing a sorted array one item at a time is very slow :
  - add(item) is n/2 on average [O(n)]
    ⇒ 1/2 + 2/2 + 3/2 + …. n/2 is $n^2/4$ [$O(n^2)$]
  - ≈ 2,500,000,000,000,000 steps for 100,000,000 items
    ⇒ 25,000,000 seconds = 289 days at 10ns per step.
- There are sorting algorithms that are much faster if you can sort whole array at once: O(n log(n))
  - ≈ 2,700,000,000 steps for 100,000,000 items
    ⇒ 27 seconds at 10ns per step.

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> colln){

    // Make space
    count=colln.size();
    data = (E[]) new Object[count];

    // Put items from collection into the data array.
    colln.toArray(data);

    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Example

- sort (34, 8, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	int moved
	34 8 64 51 32 21	
34	**8** 64 51 32 21	-
8 34	**64** 51 32 21	34
8 34 64	**51** 32 21	-
8 34 51 64	**32** 21	64
8 32 34 51 64	**21**	34, 51, 64
8 21 32 34 51 64		32, 34, 51, 64

In-place sorting

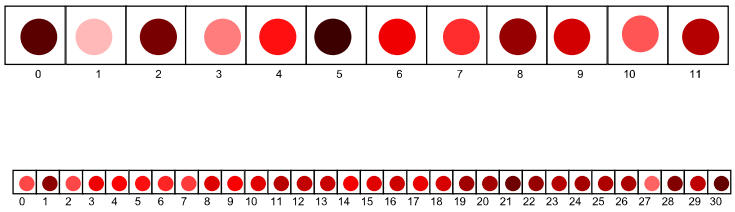
Selection Sort – Example

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

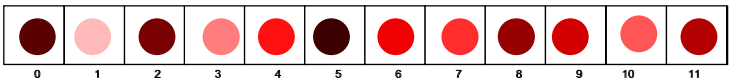
In-place sorting

Compare and Swap Sorts



- Bubble Sort (terrible)
- QuickSort (the fastest) (Divide and Conquer)

Inserting Sorts



- Insertion Sort (slow)
- Merge Sort (fast) (Divide and Conquer)

Bubble Sort

starting from the last element, swap adjacent items if they are not in ascending order

when first item is reached, the first item is the smallest

repeat the above steps for the remaining items to find the second smallest item, and so on

In-place sorting

Insertion Sort

look at elements one by one

build up sorted list by inserting the element at the correct location

Implementing Sorting Algorithms

- Could sort Lists
  - ⇒ general and flexible
  - but efficiency depends on how the List is implemented
- Could sort Arrays.
  - ⇒ less general
  - but efficiency is well defined
  - easy to convert any Collection to an array: **toArray()** method.
- Comparing items:
  - require items to be comparable (natural order)
  - provide comparator (prescribed order)
  - handle both.

Sort methods

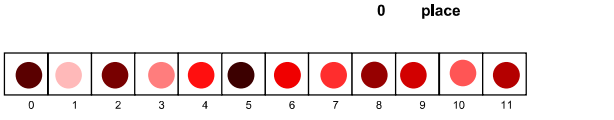
We will use:

```
public void …Sort(E[] data, int size, Comparator<E> comp)
```

- sorts first *size* elements of an array of some type, given a Comparator for that type.
- Could be used inside SortedArraySet, or standalone.
- Designing very flexible code that can be used in many different places is tricky!!

Selection Sort


```
public void selectionSort(E[ ] data, int size, Comparator<E> comp){
    // for each position, from 0 up, find the next smallest item
    // and swap it into place
    for (int place=0; place<size-1; place++){
        int minIndex = place;
        for (int sweep=place+1; sweep<size; sweep++){
            if (comp.compare(data[sweep], data[minIndex]) < 0)
                minIndex=sweep;
            }
        swap(data, place, minIndex);
    }
}
```



Bubble Sort – Example

round	(34	10	64	51	32	21)
	34	10	64	51	**32**	**21**
1	34	10	64	**51**	**21**	32
	34	10	**64**	**21**	51	32
	34	**10**	**21**	64	51	32
	34	**10**	21	64	51	32
	10	34	21	64	**51**	**32**
2	***10***	34	21	**64**	**32**	51
	10	34	**21**	**32**	64	51
	10	**34**	**21**	32	64	51
	10	***21***	34	32	64	51

Bubble Sort – Example (2)

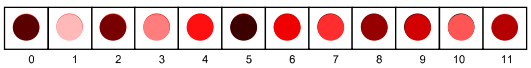
round	***10***	***21***	34	32	**64**	**51**
3	***10***	***21***	34	**32**	**51**	64
	10	***21***	**34**	**32**	51	64
	10	***21***	***32***	34	**51**	**64**
4	***10***	***21***	***32***	**34**	**51**	64
	10	***21***	***32***	***34***	**51**	**64**
5	***10***	***21***	***32***	***34***	***51***	64

bubble sort

- also called as sinking sort
- smaller values bubble their way up during the process
- need several passes through the data (how many?)
- During each pass, successive pairs of elements are compared
  - if a pair is in order, leave them as they are
  - if a pair not in order, swap

Bubble Sort

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end
    for (int top=size-1; top>0; top--){
       for (int sweep=0; sweep<top; sweep++)
          if (comp.compare(data[sweep], data[sweep+1]) >0)
              swap(data, sweep, sweep+1);
    }
}
```



Bubble Sort Analysis

- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Bubble Sort Analysis

- Efficiency
  - worst-case: **O(n^2)**
  - average case: **O(n^2),** no better than worst
- Requirements on Data
  - Needs random-access data, but can modify for files
  - Needs compare and swap
- Space Usage
  - in-place

Selection Sort Analysis

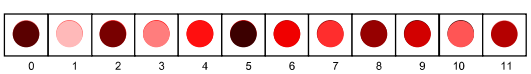
- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Selection Sort Analysis

- Efficiency
  - worst-case: **O(n^2)**
  - average case exactly the same.
- Requirements on Data
  - Needs random-access data, but easy to modify for files
  - Needs compare and swap
- Space Usage
  - in-place

Exercise: Bubble Sort Implementation

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end




}
```



Slow Sorts

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - Insertion sort is better than the others
- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.
- Solution:
  - **Must be able to compare and swap items at a distance**
  - **Must not perform redundant comparisons**

Summary

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

Readings

- [Mar07] Read 7.1, 7.2, 7.3
- [Mar13] Read 7.1, 7.2, 7.3

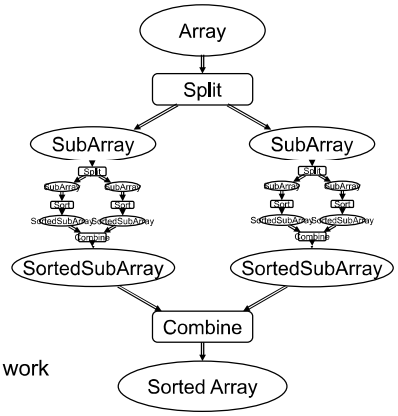
Fast Sorting

Lecture 18

Divide and Conquer Sorts

To Sort:

- Split
- Sort each part (recursive)
- Combine

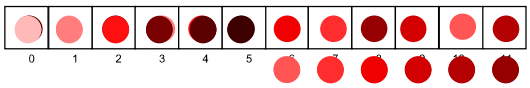


Where does the work happen?

- MergeSort:
  - split trivial
  - combine does all the work
- QuickSort:
  - split does all the work
  - combine trivial

Merge Sort

- Split the array exactly in half
- Sort each half
- "Merge" them together.



Need a temporary array

Menu

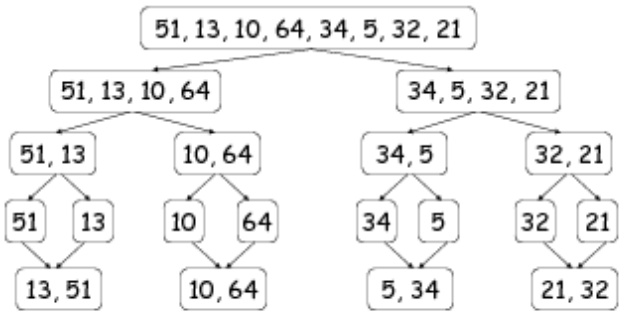
- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

51, 13, 10, 64, 34, 5, 32, 21

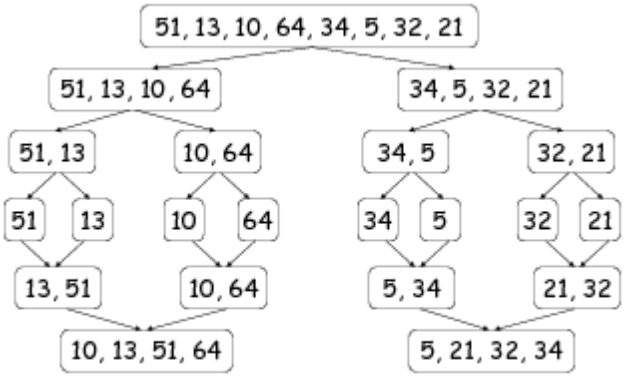
we want to sort these 8 numbers,
divide them into two halves

Slow Sorts

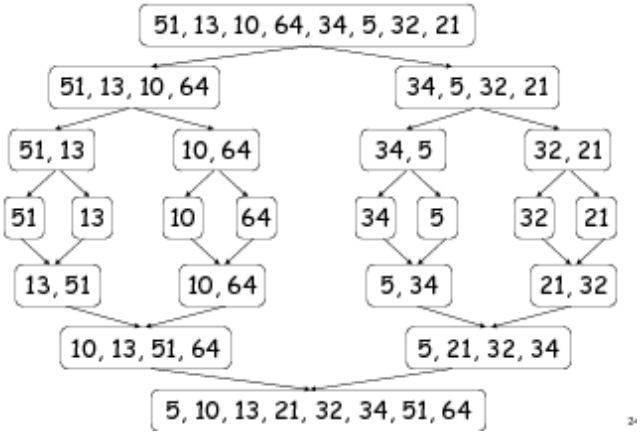
- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - $O(n^2)$
- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.
- Solution:
  - compare and swap items at a distance
  - do not perform redundant comparisons

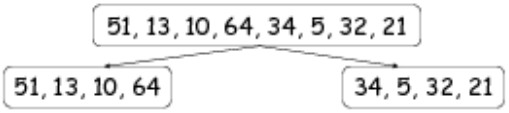


then **merge** again into sequences of 4 sorted numbers



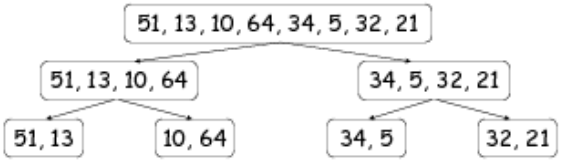
one more merge give the **final** sorted sequence



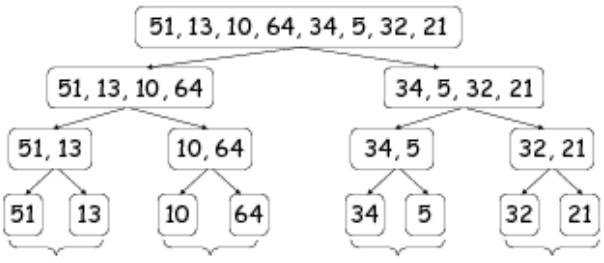


divide these 4 numbers into halves

similarly for these 4



further divide each shorter sequence ... until we get sequence with only **1** number



merge pairs of single number into a sequence of 2 sorted numbers

MergeSort


```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
    // sort items from low..high-1 using temp array
    if (high > low+1){
        int mid = (low+high)/2;
        // mid = low of upper 1/2, = high of lower half.
        mergeSort(data, temp, low, mid, comp);
        mergeSort(data, temp, mid, high, comp);
        merge(data, temp, low, mid, high, comp);
        for (int i=low; i<high; i++)   data[i]=temp[i];
    }
}
```

Sort each half

merge into temp

copy back

Mergesort – merging details

- given two sorted arrays, merge them into one sorted array
- keep track of the smallest element in each array, output the smaller of the two to a third array
- Continue until both arrays are exhausted
- If any array is exhausted first, then simply output the rest of another array

- This so-called 2-way merging can be generalized to multi-way merging

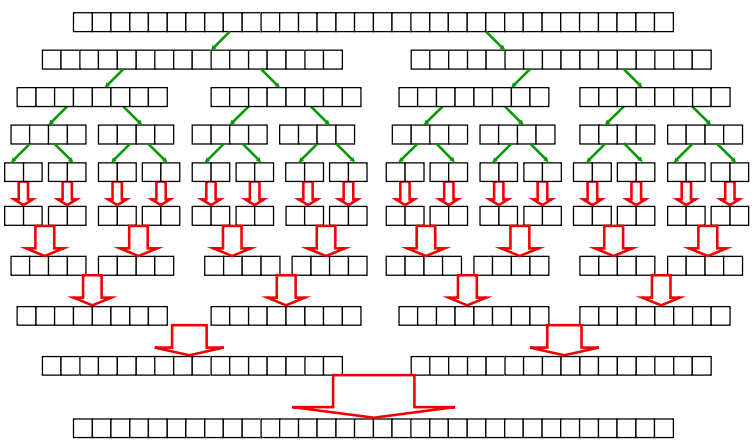
Merge


```
/** Merge from[low..mid-1] with from[mid..high-1] into  to[low..high-1.*/
private static <E> void merge(E[] from, E[] to, int low, int mid, int high,
                              Comparator<E> comp){
    int index = low;        // where we will put the item into "to"
    int indxLeft = low;     // index into the lower half of the "from" range
    int indxRight = mid;    // index into the upper half of the "from" range
    while (indxLeft<mid && indxRight < high){
        if (comp.compare(from[indxLeft], from[indxRight]) <=0)
            to[index++] = from[indxLeft++];
        else
            to[index++] = from[indxRight++];
    }
    //copy over the remainder. Note only one loop will do anything.
    while (indxLeft<mid)
        to[index++] = from[indxLeft++];
    while (indxRight<high)
        to[index++] = from[indxRight++];
}
```

Merging process in details

3 4 7 33 78 (2) 11 54 69 71 82 99 ← **2**	find the smallest of each array; compare
(3) 4 7 33 78 **11** 54 69 71 82 99 ← **2 3**	output the smaller **2**; remove 2 from the input array; find the smallest of each array; compare
(4) 7 33 78 **11** 54 69 71 82 99 ← **2 3 4**	output the smaller **3**;remove 3 from the input array; find the smallest of each array; compare
(7) 33 78 **11** 54 69 71 82 99 ← **2 3 4 7**	output the smaller **4**;remove 4 from the input array; find the smallest of each array; compare
33 78 (11) 54 69 71 82 99 ← **2 3 4 7** 11	output the smaller **7**;remove 7 from the input array; find the smallest of each array; compare

MergeSort



MergeSort

- Needs a temporary array for copying
  - create a temporary array
  - [fill with a copy of the original data.]

```
public static <E> void mergeSort(E[] data, int size,
                                 Comparator<E> comp){
    E[] other = (E[])new Object[size];
    for (int i=0; i<size; i++) other[i]=data[i];
    mergeSort(data, other, 0, size, comp);
}
```

Quicksort in process

7 4 3 9 0 8 6	find an 'i';use v = '6' to compare
l **r** **v**	use two pointers **l** & **r**, **l** scan from the left, stop when a[l] > v; r scan from right, stop when a[r]<v
0 4 3 9 **7** 8 6	swap a[l] & a[r]
0 4 3 **9** **7** 8 6	scan again from where we stop
l=r	stop again (when l >= r); i is found
0 4 3 **6** 7 8 **9**	swap a[l] with v; now every element to the left of 6 is less than 6, every element to the right of 6 is greater than 6
0 4 3 6 **7 8** 9	apply the same process to each partition
l r v **l r v**	
0 **4** 3 6 **7** 8 9	right partition sorted.
l=r	left partition stops scanning, new i found
0 **3** **4**	swap a[l] with v (3); left partition sorted
0 3 4 6 7 8 9	Done.

Exercise

- Rewrite **mergeSort** to improve its performance

QuickSort – in brief

- Divide and Conquer,
  but does its work in the "**split**" step
- It splits the array into two (possibly unequal) parts:
  - choose a "**pivot**" item
  - make sure
    - **all** items < pivot are in the **left** part
    - **all** items > pivot are in the right part
- Then (**recursively**) sorts each part

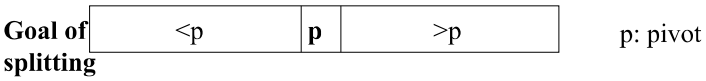
```
public static <E> void quickSort(E[] data, int size, Comparator<E> comp){
    quickSort(data, 0, size, comp);
}
```

Quicksort

- Invented by C.A.R. Hoare
- Used more widely than others

- works well for different types of data
- divide & conquer on sorting

- **O(N log N) on average**
- $O(N^2)$ worst case



Quicksort in Python

- The following quickSort code in Python from Wikipedia.

```
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

Quicksort ideas

- partition the array into two parts
- partitioning involves the selection of a[i] where the following conditions are met:
  - a[i] is in its final place in the array for some i
  - none in a[1], ... , a[i-1] is greater than a[i]
  - none in a[i+1], ... , a[r] is less than a[i]

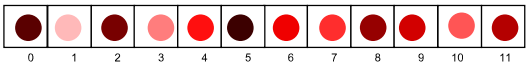
- apply quicksort recursively to each part independently

Readings

- [Mar07] Read 7.7, 7.8
- [Mar13] Read 7.6, 7.7

QuickSort in Java

```
public static <E> void quickSort(E[] data, int low, int high,
                                 Comparator<E> comp){
    if (high-low < 2)      // only one item to sort.
        return;
    else {
        // split into two parts,  mid = index of boundary
        int mid = partition(data, low, high, comp);
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```



Reflection upon Quicksort

- Quicksort makes use of ONE pivot element for partitioning.
- What if more than one pivot is used for partitioning?
- Is it feasible?

- What are the advantages of multi-pivot quicksort if feasible?

Summary

- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

Analysing Sorting Algorithms

Lecture 19

Menu

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Sorting Algorithm costs:

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - **O(n^2)**
- Merge Sort?
- Quick Sort?
  - There's no inner loop!
  - How do you analyse recursive algorithms?

MergeSort

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
  if (high > low+1){
      int mid = (low+high)/2;
      mergeSort(temp, data, low, mid, comp);
      mergeSort(temp, data, mid, high, comp);
      merge(temp, data, low, mid, high, comp);
  }
}
```

- Cost of mergeSort:
  - Three steps:
    - first recursive call: cost?
    - second recursive call: cost?
    - merge: has to copy over (high-low) items

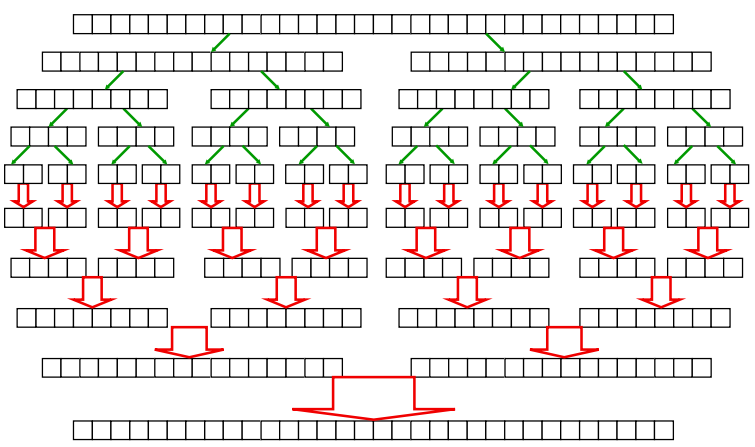
QuickSort

```
public static <E> void quickSort(E[] data, int low, int high,
                                 Comparator<E> comp){
   if (high > low +2){
      int mid = partition(data, low, high, comp);
      quickSort(data, low, mid, comp);
      quickSort(data, mid, high, comp);
   }
}
```

Cost of Quick Sort:

- three steps:
  - partition: has to compare (high-low) items
  - first recursive call
  - second recursive call

MergeSort Cost (real order)



QuickSort Cost

- If Quicksort divides the array exactly in half, then:
  - C(n) = n + 2 C(n/2)
    = n log(n) comparisons = O(n log(n))

  (best case)
- If Quicksort divides the array into 1 and n-1:
  - C(n) = n + (n-1) + (n-2) + (n-3) + … + 2 + 1
    = n(n-1)/2 comparisons = $O(n^2)$

  (worst case)
- Average case?
  - Very hard to analyse.
  - Still O(n log(n)), and very good.
- Quicksort is "in place", MergeSort is not

MergeSort Cost

- Level 1: 2 * n/2 = n
- Level 2: 4 * n/4 = n
- Level 3: 8 * n/8 = n
- Level 4: 16 * n/16 = n
- Level k: n * 1 = n
- How many levels?
- Total cost? = O( )
- n = 1,000:
  n = 1,000,000
  n = 1,000,000,000

Where have we been?

Implementing Collections:

- ArrayList: O(n) to add/remove, except at end
- Stack: O(1)
- ArraySet: O(n) (cost of searching)
- SortedArraySet O(log(n)) to search (with binary search)
  O(n) to add/remove (cost of inserting)
  $O(n^2)$ to add n items
  O(n log(n)) to initialise with n items.
  (with fast sorting)

Analysing with Recurrence Relations

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
  if (high > low+1){
      int mid = (low+high)/2;
      mergeSort(temp, data, low, mid, comp);
      mergeSort(temp, data, mid, high, comp);
      merge(temp, data, low, mid, high, comp);
  }
}
```

- Cost of mergeSort = C(n)
  - C(n) = C(n/2) + C(n/2) + n
    = 2 C(n/2) + n
- Recurrence Relation:
  - Solve by repeated substitution & find pattern
  - Solve by general method

Summary

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Solving Recurrence Relations

C(n) = 2 C(n/2) + n

= 2 [2 C(n/4) + n/2] + n

= 4 C(n/4) + 2 * n

= 4 [2 (C(n/8) + n/4] + 2 * n

= 8 C(n/8) + 3 * n

= 16 C(n/16) + 4 * n

:

$= 2^k C(n/2^k) + k * n$

when $n = 2^k$, $k = \log(n)$

= n C (1) + log(n) * n

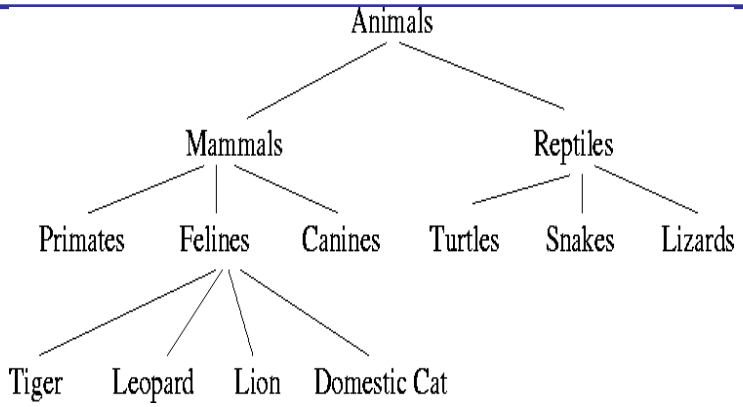
since C(1) = fixed cost

C(n) = log(n) * n

Readings

- [Mar07] Read 2.3
- [Mar13] Read 2.3

Example: Taxonomy Tree



Introduction to Trees

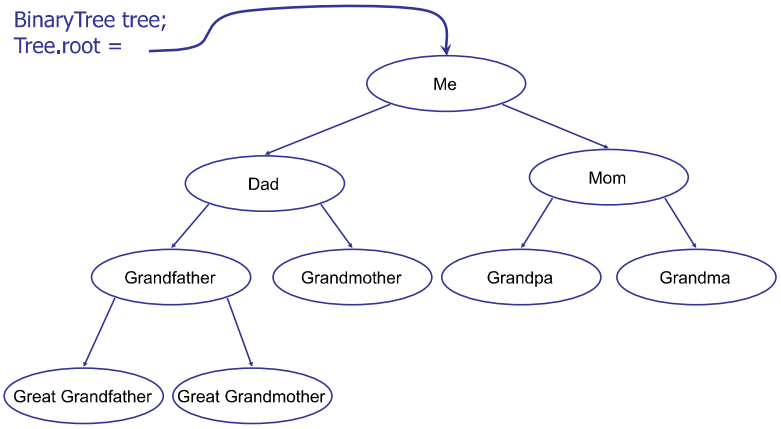
Lecture 20

Trees

- linear data structures: lists, stacks, queues
- non-linear data structures: trees

Menu

Binary Tree



Trees

- Some data are not linear (it has more structure!)
  - –Family trees
  - –Organisational charts
  - –…
- Linear implementations are sometimes inefficient
  - –Linked lists don't store such structure information
- Trees offer an alternative
  - –Representation
  - –Implementation strategy
  - –Set of algorithms

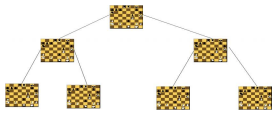
What is a tree exactly?

- Models the parent/child relationship between different elements
  - –Each child only has one parent
  - –Each parent has ? children
- From mathematics:
  - –A “directed acyclic graph” (DAG)
  - –At most one path from any one node to any other node
- Different kinds of trees exist
- Trees can be used for different purposes

- In what order do we visit elements in a tree?

Terminology

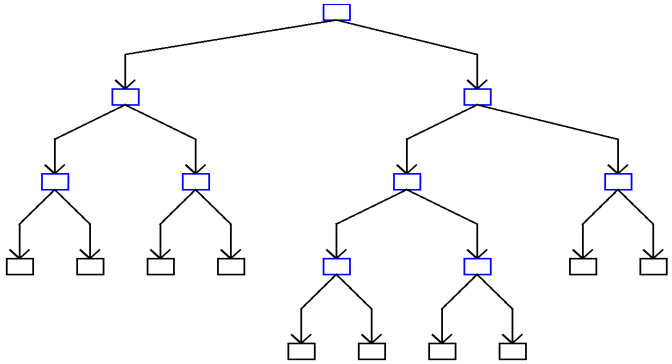
- Level:
  - –Equivalent to the “row” that a value is in
- Depth:
  - –The number of levels in the tree
- Leaf Nodes
  - –A node with no children
- Non-leaf Nodes
- Balanced:
  - –All leaf nodes are on levels n and (n+1), for some value of n (*and are grouped on the bottom level “to the left”*)



How many types of tree are there?

- Far too many:
  - –Red-Black Tree
  - –AVL Tree
  - –…
- Different types are used for different things
  - –To improve speed
  - –To improve the use of available memory
  - –To suit particular problems
- But we’ll look at three:
  - –Binary Tree
  - –General Tree (n-ary tree)
  - –AVL Tree

Binary Tree

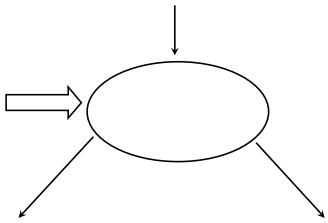


Size is limited by the depth of the tree.

Binary Tree

- Every node in the tree has a left child (or null)
- Every node in the tree has a right child (or null)
- The root of the tree is just another node

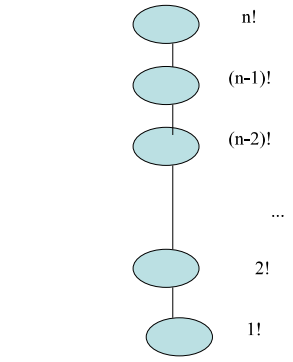
This is what you
Make a class for:



Binary tree

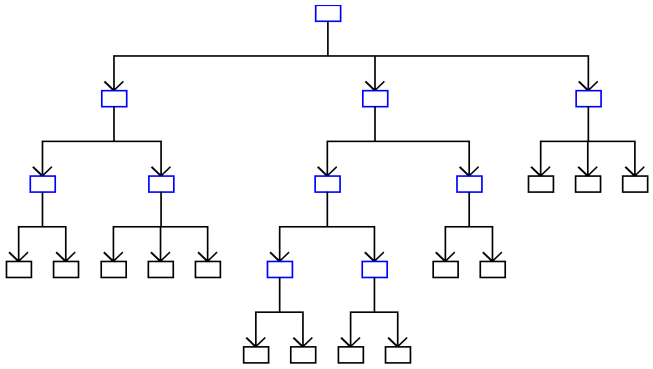
- A binary tree is either:
  - –empty or
  - –a root node together with two binary trees - left subtree & right subtree of the root

Recursion tree for n!



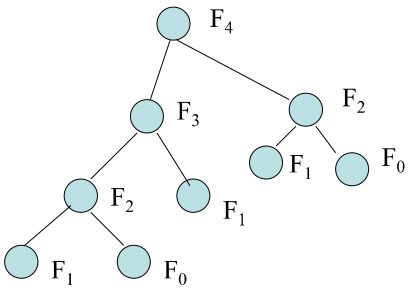
recursion tree for n!

General Trees



Question: Why is this *not* a Binary Tree?

Recursion tree for fibonacci(4)



Design Issue - Ordering of Values

- Do the children of a parent have a particular order?
  - –Does it matter which is the “first”, “second”, or “third” child?
  - –Is there an inherent ordering there?
- Are the values of the children comparable with the values of the parent?

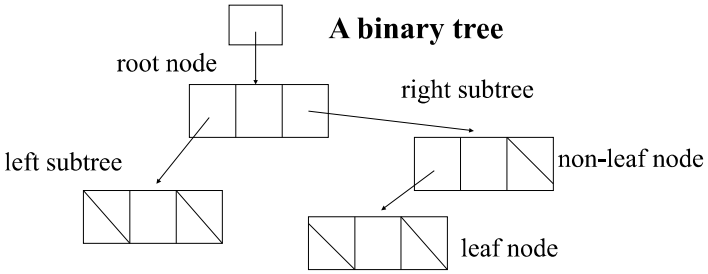
What is a tree useful for?

- Artificial Intelligence – planning, navigating, games
- Representing things:
  - –Simple file systems
  - –Class inheritance and composition
  - –Classification, e.g. taxonomy (the is-a relationship)
  - –HTML pages
  - –Parse trees for languages
  - –Essential in compilers like Java, C# etc.
  - –3D graphics (e.g. BSP trees)

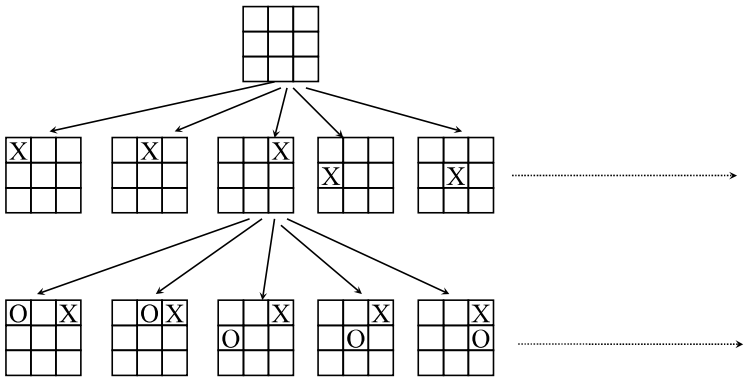
Trees and Recursion

- Recursively defined data structure
- Recursion is very natural for trees – important!
- Recursion tree
- If you don’t use recursion then use iteration

Binary tree implementation



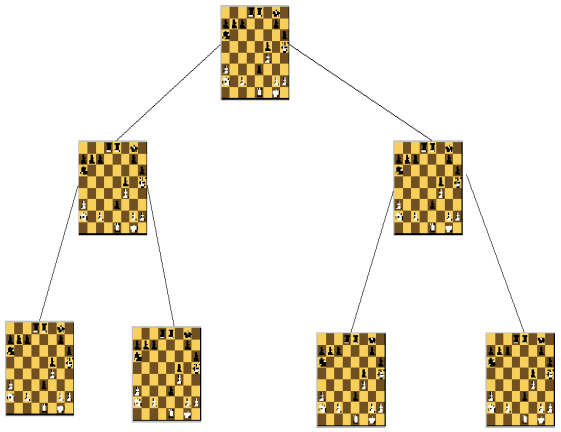
Example: Tic Tac Toe



In brief

- Data representation of tree is important
- Efficiently adding, accessing and removing data from a tree is important
- Trees can be made efficient and help you organise data
- Trees are useful in:
  - –Computer graphics
  - –Artificial Intelligence
  - –Databases
  - –…

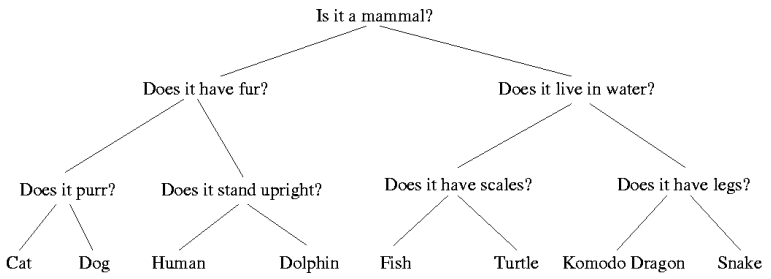
Example: Chess



Q&A

- Tree represents an efficient 1-dimensional data structure. (T or F?)
- A leaf node in a tree has no children. (T or F?)
- Binary tree has no ordering upon its sibling nodes. (T or F?)
- Name 3 applications for tree.
- Relationship among recursive calls can be expressed in what type of tree?

Example: Decision Tree



Summary

- Introduction to Trees
  - What are trees?
- Binary Tree
- General Tree
- Terminology
- Different Types of Tree
- Tree Ordering
- Trees and Recursion
- What are they useful for?
  - Tic Tac Toe example
  - Chess
  - Taxonomy Tree
  - Decision Tree

Readings

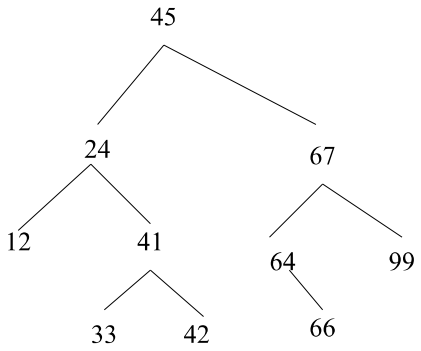
- [Mar07] Read 4.1, 4.2, 4.6
- [Mar13] Read 4.1, 4.2, 4.6

Search Lists

- a linked list with key, info, and next
- **O(N)** in average for insert, lookup, and remove

Binary Search Trees
Lecture 21

Binary Search Tree



Menu

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Binary Search Tree Definitions

- A binary search tree is a binary tree where each node has a key
- The key in the left child (if exists) of a node is less than the key in the parent
- The key in the right child (if exists) of a node is greater than the key in the parent
- The left & right subtrees of the root are again binary search trees

Tables (Maps)

- indexed container
- Associate information with a key
  - *key* is often a character string
  - *info* is any information
- E.g. a phone book
  - key is the name of a person or business
  - info is their phone number & address
- Typical Operations on Tables

```
void insert(string key, Object o);
object lookup(string key);
void remove(string key);
```

- Alternative implementations include
  - Search Lists
  - Binary Search Trees
  - Hash Tables

Binary Tree Traversal

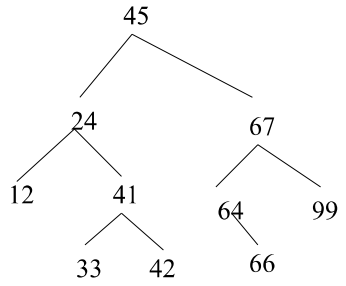
- inOrder
- preOrder
- postOrder

Binary Search Trees (BST)

- similar to a linked list, but two next pointers
- we call them *left* and *right*
- for each node n, with key k
  - n->left contains only nodes with keys < k
  - n->right contains only nodes with keys > k
- **O(log N)** in average for insert, lookup, and remove

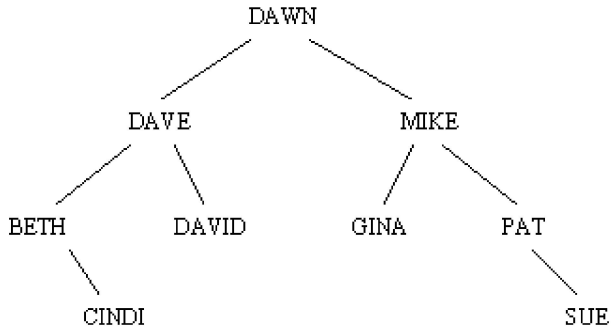
inOrder traversal: recursive

- traverse the left subtree inOrder
- process (display) the value in the node
- traverse the right subtree inOrder



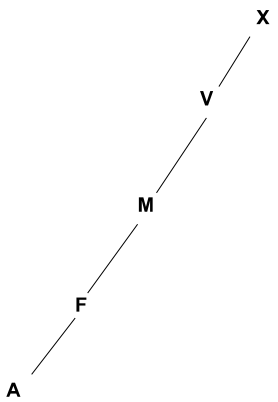
Worst Cases

- operations can degenerate to O(N) – worst case!
- degenerates to a linked list
  - when keys are inserted in ascending order
    - all keys are to the right
  - when keys are inserted in descending order
    - all keys are to the left
- ideal is mid first, then successive middles, etc.
- random order also works fairly well

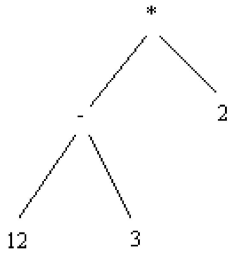


BETH, CINDI, DAVE, DAVID, DAWN, GINA, MIKE, PAT, SUE

Degenerated Tree

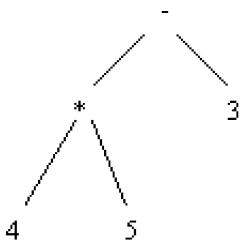


inorder traversal of the binary expression tree for (12-3)*2



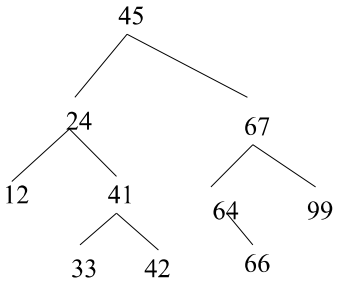
12 – 3*2

Exercise: inorder traversal of the binary expression tree for 4 * 5 - 3

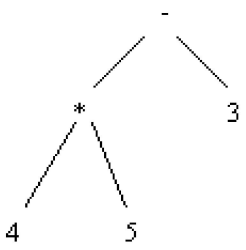


preOrder traversal: recursive

- process (display) the value in the node
- traverse the left subtree preOrder
- traverse the right subtree preOrder
- ?

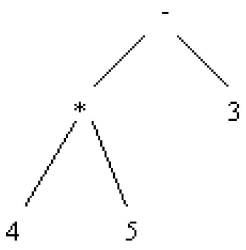


inorder traversal of the binary expression tree for 4 * 5 - 3

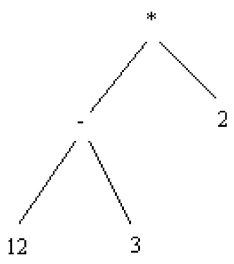


4 * 5 - 3

Exercise: preorder traversal of the binary expression tree for 4 * 5 - 3

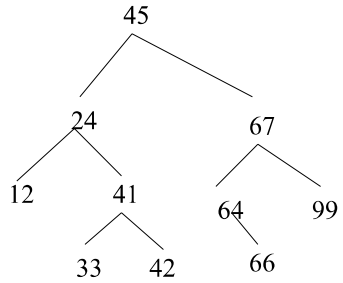


Exercise: inorder traversal of the binary expression tree for (12-3)*2

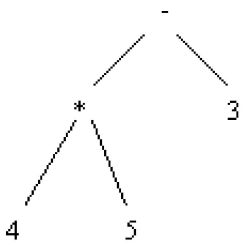


postOrder traversal: recursive

- traverse the left subtree postOrder
- traverse the right subtree postOrder
- process (display) the value in the node
- ?

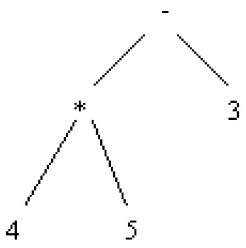


preorder traversal of the binary expression tree for 4 * 5 - 3



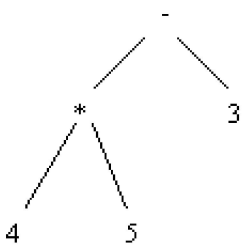
- * 4 5 3

Exercise: postorder traversal of the binary expression tree for 4 * 5 - 3



Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

postorder traversal of the binary expression tree for 4 * 5 - 3

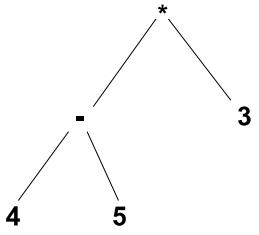


4 5 * 3 -

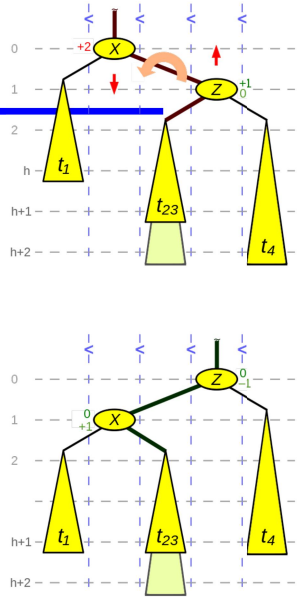
Ex: How do you construct the binary tree for 4-5*3 to be evaluated correctly in postorder?

Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

- Correct evaluation in preorder should be: * - 4 5 3
- Corresponding tree:



Simple rotation *rotate_Left(X,Z)*



Breadth-First traversal

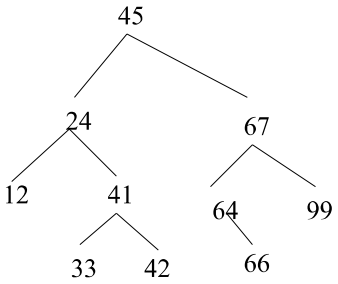
- all previous traversals are *Depth-First* traversals
- visit all the nodes at depth 0, then depth 1, etc.
- may use a **queue** to traverse across levels

AVL Trees (Adelson-Velskii and Landis)

- An AVL Tree is a form of binary search tree
- Unlike a binary search tree, the worst case scenario for a search is O(log n).
- AVL data structure achieves this property by placing restrictions on the difference in height between the sub-trees of a given node - height balanced to within 1
- and re-balancing the tree if it violates these restrictions.

Breadth-First traversal

- ?



AVL Tree Balance Requirements

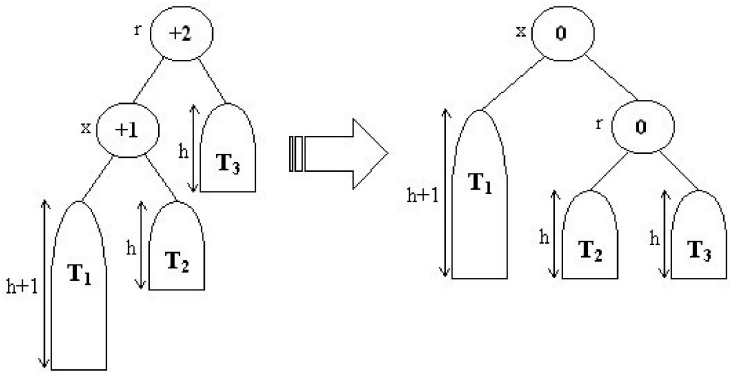
- A node is only allowed to possess one of three possible states:
- **Left-High (balance factor -1)**
  The left-sub tree is one level taller than the right-sub tree
- **Balanced (balance factor 0)**
  The left and right sub-trees both have the same heights
- **Right-High (balance factor +1)**
  The right sub-tree is one level taller than the left-sub tree

- If the balance of a node becomes -2 or +2 it will require re-balancing.
- This is achieved by performing a **rotation** about this node
- **Rotation does not break the existing properties for a search tree**

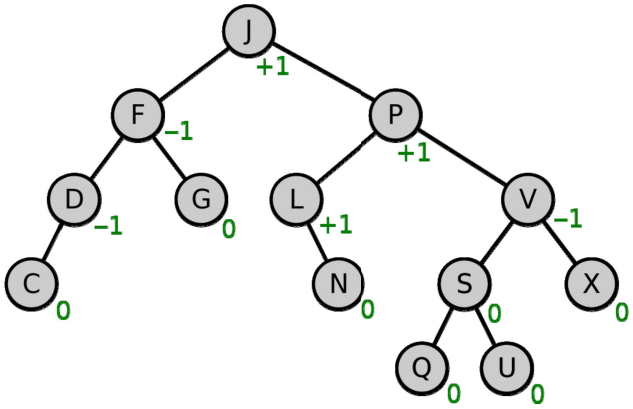
Balanced Search Trees

- use **rotations** to ensure tree is always 'full'
- prevents degenerative cases mentioned above
- truely O(log N) worst case for insert, lookup, remove
- insertion/removal takes more time
- but lookup is faster
- trickier to code correctly

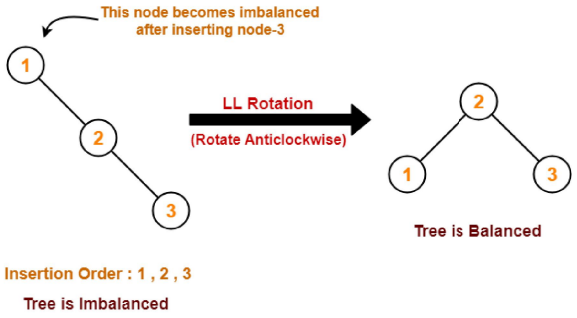
AVL Rotation - RR



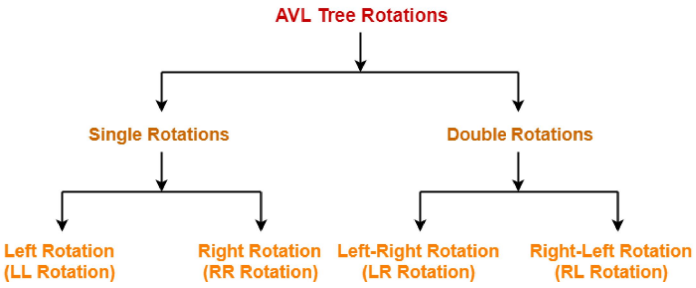
AVL tree with balance factors



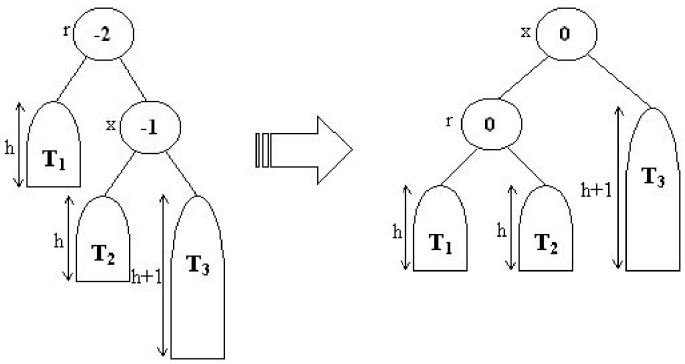
AVL Rotation - LL



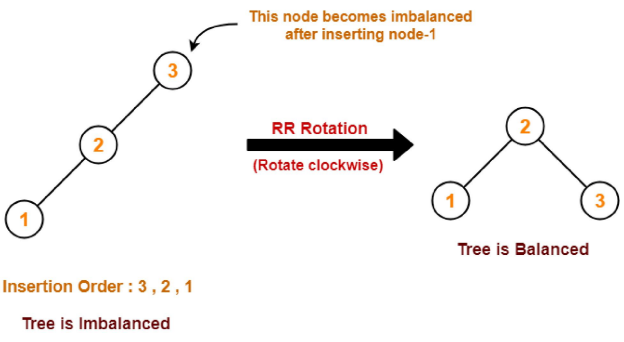
AVL Tree Rotations



AVL Rotation - LL



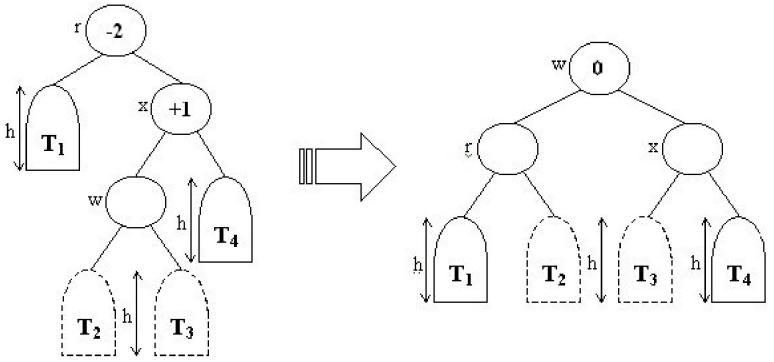
AVL Rotation - RR



AVL Tree Insertion

- AVL requires two passes for insertion:
- one pass down tree (to determine insertion)
- one pass back up to update heights and rebalance

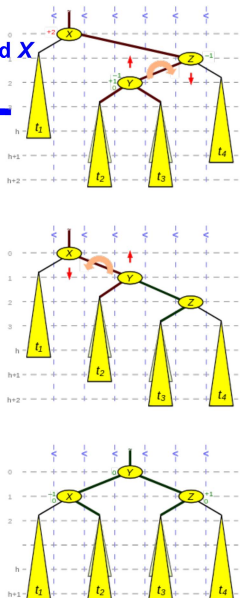
AVL Rotation - RL



AVL Tree Insertion

- Animation showing the insertion of several elements into an AVL tree. It includes left, right, left-right and right-left rotations.

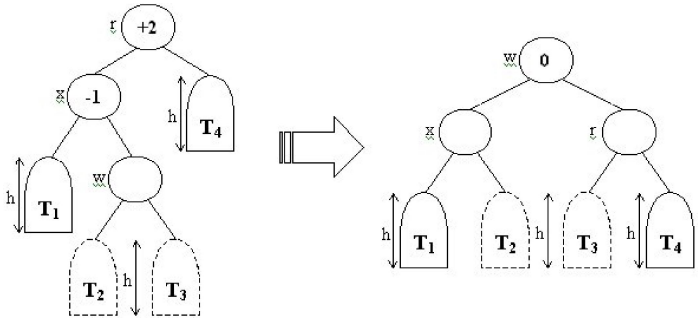
AVL Double rotation *rotate_RightLeft(X,Z)* = *rotate_Right* around *Z* followed by *rotate_Left* around *X*



AVL Time complexity in big O notation

Algorithm	Average	Worst case
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)
Space	O(*n*)	O(*n*)

AVL Rotation - LR



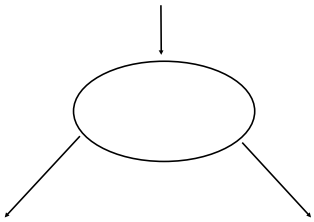
Summary

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Readings

- [Mar07] Read 4.2, 4.3, 4.4, 4.8, 9.6
- [Mar13] Read 4.2, 4.3, 4.4, 4.8

What does the Binary Tree ADT/class need?



Implementing Trees

Lecture 22

Binary Tree

- Each node contains a value
- Each node has a left child and a right child (may be null)

```
public class BinaryTree<V> {

   private V value;
   private BinaryTree<V> left;
   private BinaryTree<V> right;

   /**Creates a new tree with one node
        (a root node) containing the specified value  */
   public BinaryTree(V value) {
     this.value = value;
   }
```


Menu

- Abstract Data Type (ADT) for Trees
- Implementing Binary Trees: issues & considerations, data structure?
- Implementing General Trees: issues & considerations, data structure?

- Applications for Trees
- Tree Traversal

Get and Set

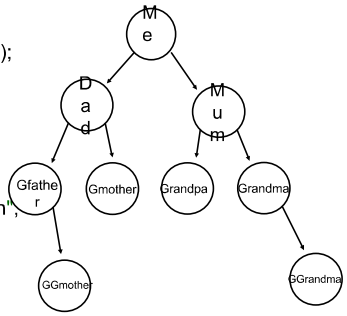
```
  public V getValue() {
      return value;
   }
   public BinaryTree<V> getLeft() {
      return left;
   }
   public BinaryTree<V> getRight() {
      return right;
   }
   public void setValue(V val) {
       value = val;
   }
   public void setLeft(BinaryTree<V> tree) {
       left = tree;
   }
   public void setRight(BinaryTree<V> tree) {
       right = tree;
   }
```


Binary Tree ADT

```
public interface BinaryTree<T> {
    BinaryTree<T> BinaryTree(T value);

    BinaryTree<T> getLeft();
    BinaryTree<T> getRight();
    void setLeft(BinaryTree<T> subtree);
    void setRight(BinaryTree<T> subtree);

    BinaryTree<T> find(T value);
    boolean contains(T value);

    boolean isEmpty();
    boolean isLeaf();
    int size();
    }
```

Using BinaryTree

```
BinaryTree<String> ma = myTree;
   while(ma.getRight()!=null)   ma = ma.getRight();

BinaryTree<String> pa = myTree;
    while(pa.getLeft()!=null)   pa = pa.getLeft();

System.out.format(
       "paternal ansc = %s, maternal ansc = %s\n\n",
       pa.getValue(), ma.getValue());
```



What would be the results of execution?

paternal ans = Gfather, maternal ans = GGrandma

isLeaf and find

```
public boolean isLeaf() {
    return (right == null) && (left == null);
 }

 public BinaryTree<V> find(V val) {
    if (value.equals(val))      return this;
    if (left != null) {
        BinaryTree<V> ans = left.find(val);
        if (ans != null)        return ans;
    }
    if (right != null) {
        BinaryTree<V> ans = right.find(val);
        if (ans != null)        return ans;
    }
    return null;
  }
}
```


Using BinaryTree

```
public static void printAll(BinaryTree<String> tree, String indent){
   System.out.println(indent+ tree.getValue());
   if (tree.getLeft()!=null)
     printAll(tree.getLeft(), indent+"  " );
   if (tree.getRight()!=null)
     printAll(tree.getRight(), indent+"  " );
}
```

What traversal scheme is this?

PreOrder traversal

Using BinaryTree

- A method that constructs a tree

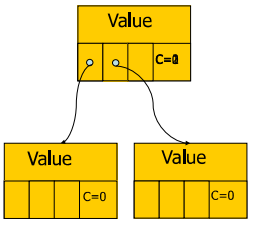
```
public static void main(String[] args){
   BinaryTree<String> myTree = new BinaryTree<String>("Me");
   myTree.setLeft(new BinaryTree<String>("Dad"));
   myTree.setRight(new BinaryTree<String>("Mum"));
   myTree.getLeft().setLeft(new BinaryTree<String>("Gfather"));
   myTree.getLeft().setRight(new BinaryTree<String>("Gmother"));
   myTree.getRight().setLeft(new BinaryTree<String>("Grandpa"));
   myTree.getRight().setRight(new BinaryTree<String>("Grandma"));
   myTree.getRight().getRight().setRight(
            new BinaryTree<String>("GGrandma"));

   BinaryTree<String> gf = myTree.find("Gfather");
   if (gf!=null)
      gf.setRight(new BinaryTree<String>("GGmother"));
}
```

Ex: Show the final tree contents after the above program execution.

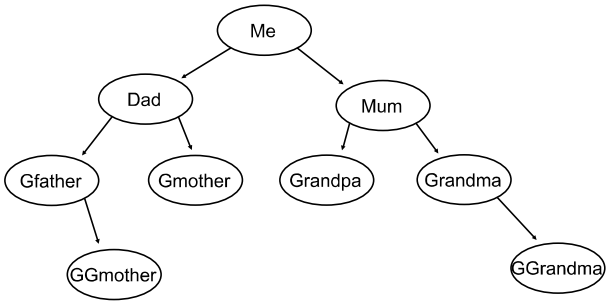
General Tree

- A node in the tree can have any number of children
- We keep the children in the order that they were added.

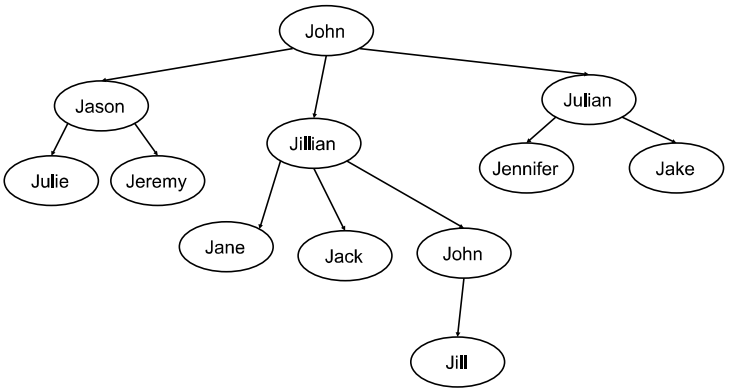
```
public class GeneralTree<V> {

   private V value;
   private List< GeneralTree<V> > children;

   public GeneralTree(V value) {
      this.value = value;
      this.children = new ArrayList< GeneralTree<V> >();
   }
...
}
```



myTree



The Tree



Get

- Children are ordered, so can ask for the *i* th child

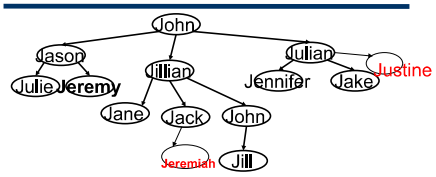
```
public V getValue() {
   return value;
}

public List< GeneralTree<V> > getChildren() {
   return children;
}

public GeneralTree<V> getChild(int i) {
   if (i>=0 && i < children.size())
      return children.get(i);
   else
      return null;
}
```


More Adding

```
GeneralTree<String> thrd = mytree.getChildren().get(2);
thrd.addChild(new GeneralTree<String>("Justine"));

GeneralTree<String> gc = mytree.find("Jack");
if (gc!=null)
    gc.addChild(new GeneralTree<String>("Jeremiah"));

System.out.println ("2nd child of 1st child: "+
                    mytree.getChild(0).getChild(1).getValue());
```

Jeremy

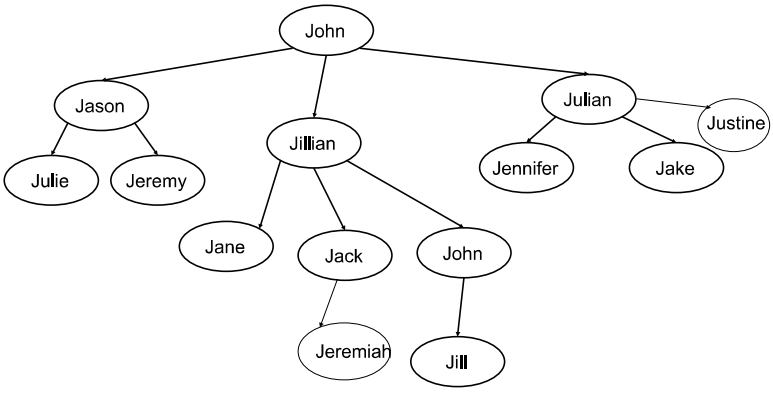
mytree = ?



add and find

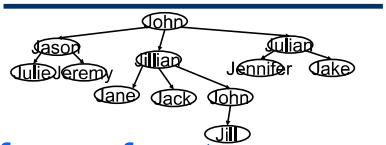
```
public void addChild(GeneralTree<V> child) {
    children.add(child);
 }
 public void addChild(int i, GeneralTree<V> child) {
    children.add(i, child);
 }

public GeneralTree<V> find(V val) {
    if (value.equals(val))       return this;
    for(GeneralTree<V> child : children){
        GeneralTree<V> ans = child.find(val);
        if (ans != null)          return ans;
    }
    return null;
}
```


The Tree



Using General Tree

```
public static void main(String[] args){
 GeneralTree<String> mytree = new GeneralTree<String>("John");
    mytree.addChild(new GeneralTree<String>("Jason"));
    mytree.addChild(new GeneralTree<String>("Jillian"));
    mytree.addChild(new GeneralTree<String>("Julian"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Julie"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Jeremy"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jane"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jack"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("John"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jennifer"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jake"));
    mytree.getChildren().get(1).getChildren().get(2).addChild(new
            GeneralTree<String>("Jill"));
}
```



Ex. Draw the final form of mytree

printAll

```
printAll(mytree, "");

public static void printAll(GeneralTree<String> tree, String indent){
    System.out.println(indent+ tree.getValue());
    for(GeneralTree<String> child : tree.getChildren())
      printAll(child, indent+"  " );
  }
```

What traversal scheme is this?

PreOrder traversal

Summary

- Investigated the design and implementation of binary tree and general tree
  - Issues
  - Data Structure
- Tree class design and implementation is quite simple… if you get the ideas of recursion
- Every algorithm can be expressed iteratively or recursively
- One way is usually much better than the other!
  - In trees, recursion is normally nicer than iteration (but not always)

Comparison among various search algorithms – Linear Search

	name	number
0	Parker	12345
1	Davis	43534
2	Harris	32452
3	Corea	46532
4	Hancock	96562
5	Brecker	37811
6	(empty)	
…	…	
N-1	Marsalis	54323

Linear Search = O(N)

Lecture 23
Hashing and Hash Tables

Comparison among various search algorithms – Binary Search

	name	number
0	Brecker	37811
1	Corea	46532
2	Davis	43534
3	Hancock	96562
4	Harris	32452
5	Marsalis	54323
6	Parker	12345
7	(empty)	
…	…	
N-1	(empty)	

Binary Search = O(log(N))

Menu

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Comparison among various search algorithms – Hash Table

name	hash
Brecker	6
Corea	2
Davis	2
Hancock	12
Harris	12
Marsalis	2
Parker	8

Hash code	name/number
0	
1	
2	Corea/46532, Davis/43534, Marsalis/54323
3	
4	
5	
6	Brecker/37811
7	
8	Parker/12345
9	
10	
11	
12	Hancock/96562, Harris/32452

Hash = O(1) if no collision
hash function is simply the sum of ASCII codes of characters in a name (considered all in lowercase) computed mod N=13.

Hash Tables

- another kind of Table
- **O(1)** in average for insert, lookup, and remove
- use an array named T of *capacity* N
- define a hash function that returns an integer int **H(string key)**
- must return an integer between 0 and N-1
- store the key and info at T[H(key)]
- H() must always return the same integer for a given key

Hash Functions

- a good hash function has the following characteristics:

- avoids collisions
- spreads keys evenly in the array
- inexpensive to compute - must be O(1)

Table Size

- Table size is usually *prime* to avoid bias
- overly large table size means wasted space
- overly small table size means more collisions
- what happens as table size approaches 1?

Hash Function for Signed Integer Keys

- remainder after division by table length

```
int hash(int key, int N) {
return abs(key) % N;
}
```

- if keys are positive, you can eliminate the abs

What is a Hash Function?

- A **hash function** is any well-defined procedure or mathematical function for turning data into an index into an array.
- The values returned by a hash function are called **hash values** or simply **hashes.**
- A hash function H is a transformation that
  - takes a variable-size input k and
  - returns a fixed-size string (or int), which is called the ***hash value*** h (that is, h = H(k))

- In general, a hash function may map several different keys to the same hash value.

Hash Functions for Strings

- must be careful to cover range from 0 through capacity-1
- some poor choices
  - summing all the ASCII codes
  - multiplying the ASCII codes
- important insight
  - letters and digits fall in range 0101 and 0172 octal
  - so all useful information is in lowest 6 bits
- hash(s) is O(1)

Example of a Modular Hash Function

- H(k) = **k mod m** (or **k % m**)
- message1 = '723416'
- hash function = modulo 11
- Hash value$_1$ = (7+2+3+4+1+6) mod 11 = 1

- message2 = 'test' = ASCII '74', '65', '73', '74'
- Hash value$_2$ = (74+65+73+74) mod 11 = 0
- another hash function example: **a*k mod m**

Dealing with Collisions

- ***open addressing*** - collision resolution
- key/value pairs are stored in array slots
- ***linear probing***
  - hash(k, i) = (hash1(k) + i) mod N
  - increment hash value by a constant, 1, until free slot is found
  - simplest to implement
  - leads to *primary clustering*
- ***quadratic probing***
  - hash(k, i) = (hash1(k) + c_1*i + c_2*i*i) mod N
  - leads to *secondary clustering*
- ***double hashing***
  - hash(k, i) = (hash1(k) + i*hash2(k)) mod N
    - avoids clustering

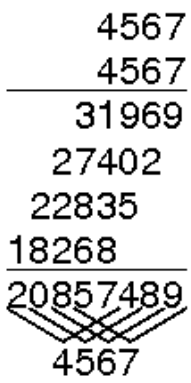
Hash Functions for Integer Keys

- *Mid-square* method
- Squaring the key value first, and then takes out the middle *r* bits of the result, giving a value in the range 0 to 2^r-1.
- This works well because most or all bits of the key value contribute to the result

Dealing with Collisions

- *separate chaining*
- each array slot is a SearchList
- never gets 'full'
- deletions are not a problem

Mid-square Method



Dealing with A Full Table

- allocate a larger hash table
- rehash each from the smaller into the larger
- delete the smaller

Hash Functions for String Keys

- This function takes a string as input. It processes the string 4 bytes at a time, and interprets each of the four-byte chunks as a single long integer value.
- The integer values for the 4-byte chunks are added together.
- The resulting sum is converted to the range 0 to *M*−1 using the modulus operator.
- There is nothing special about using 4 characters at a time. Other choices could be made.

Why hash table can give us O(1) performance?

- It appears most search mechanisms have performance at O(N) or O(logN)
- So why hash table can give us best performance?
- Where does the magic come from?

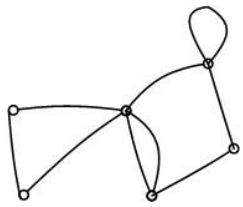
Summary

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Readings

- [Mar07] Read 5.1-5.4, 5.6
- [Mar13] Read 5.1-5.4, 5.6

Give the number of vertices and the number of edges of the following graph:



- $|V| = 6$;
- $|E| = 9$.

Introduction to Graph Theory
Lecture 24

Incidence, adjacency and neighbors

- Two vertices are **adjacent** if they are joined by an edge.
- Adjacent vertices are said to be **neighbors.**
- The edge which joins vertices is said to be **incident** to them.

Menu

- Basic definitions of graph theory
- Properties of graphs
- Paths
- Trees
- Digraphs and their applications, network flows

Multiple edges, loops and simple graphs

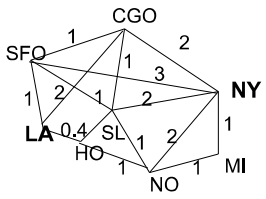
- Two or more edges joining the same pair of vertices are **multiple edges.**
- An edge joining a vertex to itself is called a **loop.**
- A graph containing no multiple edges or loops is called a **simple graph**

Definitions

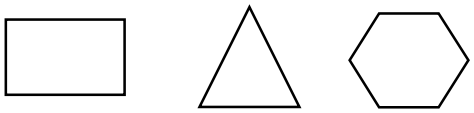
- A **graph** G consist of :
  - a *finite* set of **vertices** $V(G)$, which cannot be empty,
  - and a *finite* set of **edges** $E(G)$, which connect pairs of vertices.
- The number of vertices in G is called the **order** of G, denoted by $|V|$.

Why study graph theory?

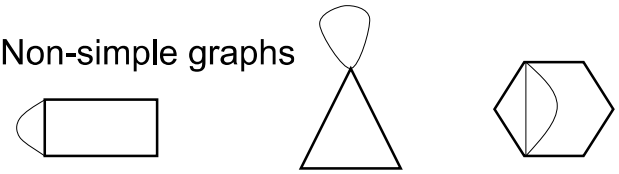
- Routing problem: find the minimal delay path from LA to NY.



Simple graph: examples

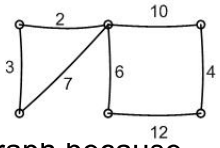


- Non-simple graphs



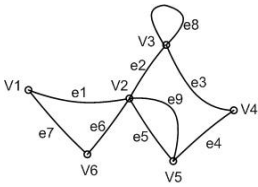
Weighted graphs

- A **weighted graph** has a number assigned to each of its edges, called its **weight.**
- The weight can be used to represent distances, capacities or costs.
- Is the following weighted graph a simple graph? Justify your answer



- The weighted graph is a simple graph because it has no multiple edges or loops

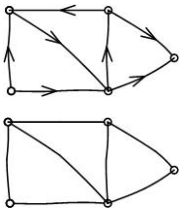
In the following graph:
Identify the neighbours of *V*4
Identify the edge incident to *V*3 and *V*4
Identify multiple edges
Identify the loop



- The neighbours of *V*4 are: *V*3 and *V*5
- The edge incident to *V*3 and *V*4 is: *e*3
- *e*5 and *e*9 are multiple edges
- *e*8 is a loop

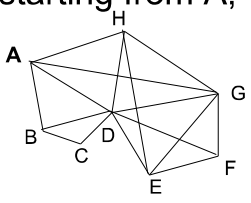
Digraphs

- A **digraph** is a *directed* graph, a graph where instead of edges we have directed edges with arrows (**arcs**) indicating the direction of flow.
- Sketch the underlying graph of the digraph:



Why study graph theory?

- There are many engineering or computer science related problems that can be modelled using 'graphs'
- For example, **travelling salesman problem**: find the minimal cost path to cover all the cities A-H, starting from A, ending at A



Degree

- For a digraph we get

$$\sum_i d_-(V_i) = |A|$$

$$\sum_i d_+(V_i) = |A|$$

- where |A| is the number of arcs.

Degree

- The *number of times* edges are incident to a vertex *V* is called its **degree**, denoted by *d*(*V*).
- The **degree sequence** of a graph consists of the degrees of the vertices written in non-*increasing order*, with repeats where necessary.
- The sum of the values of the degrees, *d*(*V*), over all the vertices of a simple graph is twice the number of edges:

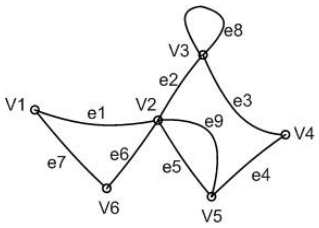
$$\sum_i d(V_i) = 2|E|$$

- Why?

Subgraphs

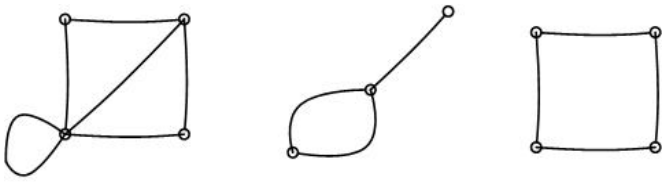
- A **subgraph** of *G* is a graph, *H*, whose vertex set is a subset of *G*'s vertex set, and whose edge set is a subset of the edge set of *G*.
- If a subgraph *H* of *G* spans all of the vertices of *G*, i.e. *V*(*H*) = *V*(*G*), then *H* is called a **spanning subgraph** of *G.*

Give the degrees of the vertices *V*1 and *V*3 of the graph of



- *d*(*V*1) = 2 and *d*(*V*3) = 4

For the graph (a) which of the subgraphs (b) and (c) is a spanning subgraph?



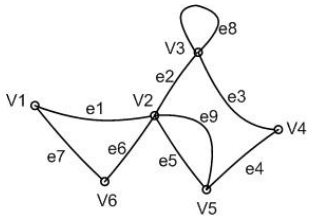
- Subgraph (c) is a spanning subgraph of graph (a).

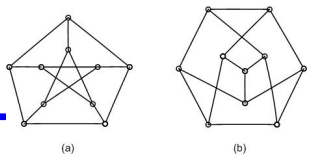
Degree

- A vertex of a digraph has an in-degree of *d*-(*V*) and an out-degree *d*+(*V*).

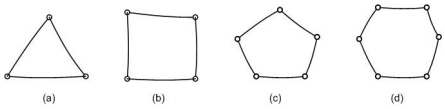
Self test

- 1. Write down the vertex set and edge set of the graph in:

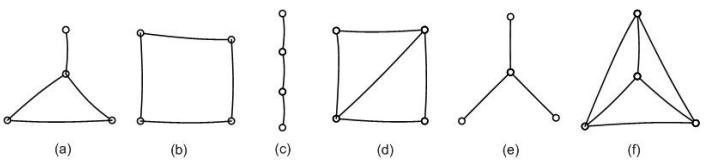




- 2. Which graphs below are subgraphs of those shown above.



- 3. Write down the degree sequence in the graphs below. Verify that the sum of the values of the degrees are equal to twice the number of edges in the graph.



Summary

- Definitions of **graphs: vertices, edges, order**
- Definitions of: **multiple edges, loops**
- Definitions of: **simple graphs**
- **Digraph**: directed graph
- **Weighted graphs**
- The number of times edges are incident to a vertex *V* is called its **degree**

Summary

- The sum of the values of the degrees, $d(V)$, over all the vertices of a simple graph is twice the number of edges: $\sum_i d(V_i) = 2|E|$
- Definitions of: **subgraphs, spanning subgraphs**

Readings

- [Mar07] Read 9.1
- [Mar13] Read 9.1

1. Answers:

- The vertex set is {V1,V2,V3,V4,V5,V6},
- The edge set is {e1,e2,e3,e4,e5,e6,e7,e8, e9}.

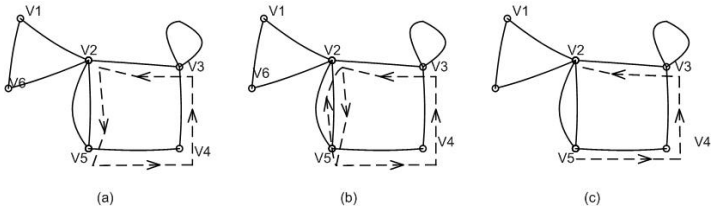
Answers

- 2. (c), (d)

- **Answers**

- 3. (a) (3, 2, 2, 1);
  (b) (2, 2, 2, 2);
  (c) (2, 2, 1, 1);
  (d) (3, 3, 2, 2);
  (e) (3, 1, 1, 1);
  (f) (3, 3, 3, 3);

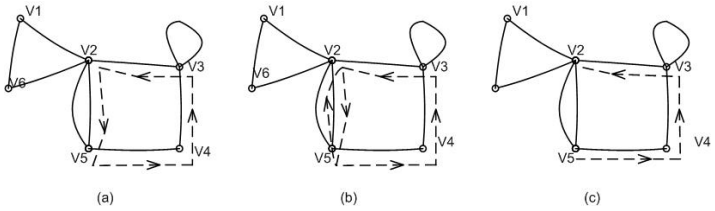
Example: Identify whether a path is marked on the graph in each case:



- Solution: (c) is a path, length?

Introduction to Graph Theory
Lecture 25

Example: Identify whether a trail, path or circuit is marked on the graph in each case:



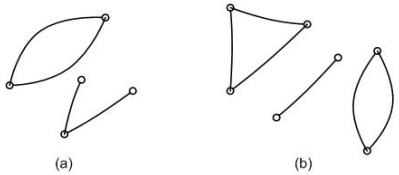
- Solution: (a) circuit (b) trail (c) path

Menu

- Paths
- Connected graphs
- Incidence matrix and adjacency matrix of a graph

Connected graphs

- A graph *G* is **connected** if there is a path from any one of its vertices to any other vertex**.**
- A **disconnected** graph is said to be made up of **components.**
- Example 5:
- How many components do the following disconnected graphs have?

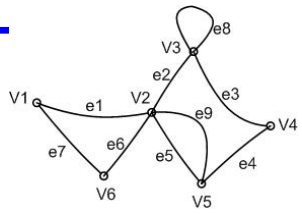


- Solution:
  (a) Two components (b) Three components

Walks, paths and circuits

- A sequence of edges of the form
  V_sV_i, V_iV_j, V_jV_k, V_kV_l, V_lV_t is a **walk** from V_s to V_t.
- If these edges are distinct then the walk is called a **trail**, and
- if the vertices are also distinct then the walk is called a **path.**
- A walk or trail is **closed** if $V_s = V_t$.
- A closed walk in which all the vertices are distinct except V_s and V_t, is called a **cycle** or a **circuit.**
- The number of edges in a walk is called its **length.**

Give the adjacency matrix of the graph below:



- The adjacency matrix for the graph is given by

$$\begin{array}{c} \\ V1 \\ V2 \\ V3 \\ V4 \\ V5 \\ V6 \end{array} \begin{array}{c} \begin{array}{cccccc} V1 & V2 & V3 & V4 & V5 & V6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

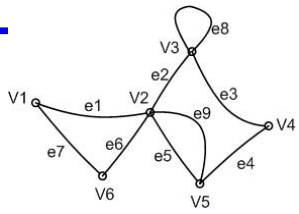
Matrix representation of a graph: the incidence matrix

- The **incidence matrix** of a graph G is a $|V| \times |E|$ matrix $\boldsymbol{A}$.
- The element a_{ij} =
- the *number of times* that vertex V_i is incident with the edge e_j

Summary

- Definitions of **paths**
- Definitions of **connected graphs**
  Definitions of **incidence matrix** and **adjacency matrix** of a graph

Give the incidence matrix of the graph below:



- The incidence matrix for the graph is given by

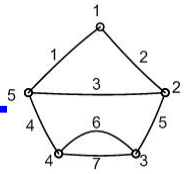
$$\begin{array}{c} \\ V1 \\ V2 \\ V3 \\ V4 \\ V5 \\ V6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} e1 & e2 & e3 & e4 & e5 & e6 & e7 & e8 & e9 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Q. How would you design data structure for graphs? What type of data structure can we use to store graphs?

Matrix representation of a graph: the adjacency matrix

- The **adjacency matrix** of a graph G is a $|V| \times |V|$ matrix $\boldsymbol{A}$.
- The element a_{ij} =
- the *number of edges* joining V_i and V_j

Answers



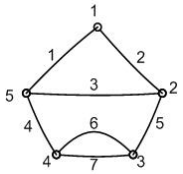
- 1. The incidence matrix is

	1	2	3	4	5	6	7
1	1	1	0	0	0	0	0
2	0	1	1	0	1	0	0
3	0	0	0	0	1	1	1
4	0	0	0	1	0	1	1
5	1	0	1	1	0	0	0

The adjacency matrix is

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	0	1
3	0	1	0	2	0
4	0	0	2	0	1
5	1	1	0	1	0

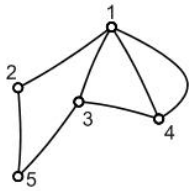
- **Self test**
- 1. Write down the adjacency and incidence matrices of the graph below.



Answers

2.

	1	2	3	4	5
1	0	1	1	2	0
2	1	0	0	0	1
3	1	0	0	1	1
4	2	0	1	0	0
5	0	1	1	0	0



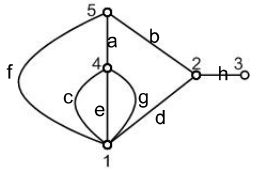
- **Self test**
- 2. Draw the graph whose adjacency matrix is given in (a)

0	1	1	2	0
1	0	0	0	1
1	0	0	1	1
2	0	1	0	0
0	1	1	0	0

(a)

	a	b	c	d	e	f	g	h
1	0	0	1	1	1	1	1	0
2	0	1	0	1	0	0	0	1
3	0	0	0	0	0	0	0	1
4	1	0	1	0	1	0	1	0
5	1	1	0	0	0	1	0	0

3.



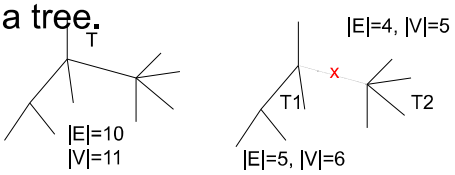
- **Self test**
- 3. Draw the graph whose incidence matrix is given in (b)

0	0	1	1	1	1	1	0
0	1	0	1	0	0	0	1
0	0	0	0	0	0	0	1
1	0	1	0	1	0	1	0
1	1	0	0	0	1	0	0

(b)

Tree properties

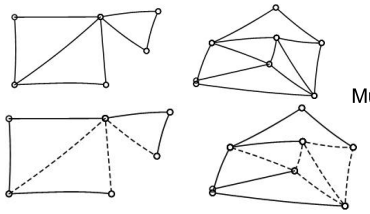
- If a tree *T* has at least two vertices then it has the following properties:
- There is exactly one path from any vertex V_i in *T* to any other vertex V_j
- The graph obtained from tree *T* by removing any edge has two components, each of which is a tree.
- $|E| = |V| - 1$



Introduction to Graph Theory
Lecture 26

Spanning trees

- A **spanning tree** of a graph *G* is
  - a tree *T*
  - a spanning subgraph of *G*.
  - That is, *T* has the same vertex set as *G*.
- Example 2 Identify a spanning tree for each of the following graphs:



Hint: remove any edge which forms a cycle

Multiple spanning trees may exist

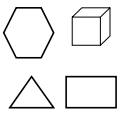
Menu

- Trees and forests
- Spanning trees
- Minimum spanning tree
- Greedy algorithm for determining a minimum spanning tree
- Shortest path problem

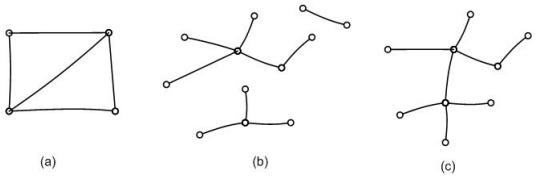
Given a graph G: How to draw a spanning tree?

- Take any vertex of *G* as an initial partial tree.
- Add edges one by one so each new edge joins a new vertex to the partial tree.
- When to stop?
- If there are *n* vertices in the graph *G* then the spanning tree will have *n* vertices and *n*-1 edges.

Trees



- A **tree** is a connected graph with no cycles.
- A **forest** is a graph with no cycles and it may or may not be connected
- Example 1: Identify which of the following graphs are trees or forests.



- Solution: (b) A forest (c) A tree

The gr$$dy algorithm for the minimum spanning tree

- Choose any start vertex to form the initial partial tree (*Vi*)
- Add the **chapst** edge, *Ei*, to a new vertex to form a new partial tree
- Repeat step 2 until all vertices have been included in the tree

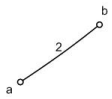
- Why is it *greedy*?

Minimum spanning tree

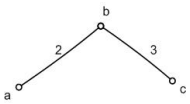
- Suppose we have a group of offices which need to be connected by a network of communication lines.
- The offices may communicate with each other directly or through another office.
- Condition: there exists one path between any two vertices.
-
- In order to decide on which offices to build links between we firstly work out the cost of all possible connections.
- This will give us a weighted complete graph as shown next.
- The **minimum spanning tree** is then the spanning tree that has the minimum cost among all spanning trees.

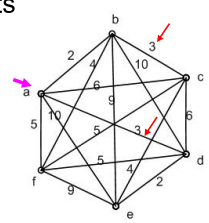
Find the minimum spanning tree for the graph representing communication links between offices as shown below.

- Start with any vertex, in this case choose the one marked *a*.
- Add the edge *ab* which is the cheapest edge of those adjacent to *a*.



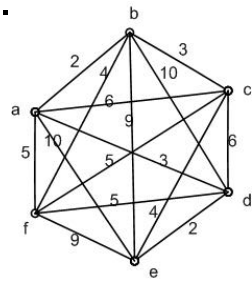
- Looking for the cheapest edge from among those incident to a or b, we find edges *bc* and *ad*, both costing 3, and that no other available edge costs less. We can choose either *bc* or *ad*. Arbitrarily we choose *bc*.





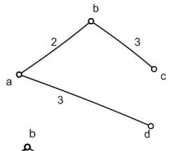
Minimum spanning tree

- A weighted complete graph.
- The vertices represent offices and the edges possible communication links.
- The weights on the edges represent the cost of construction of the link.

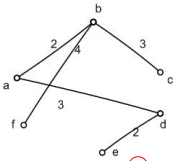


Find the minimum spanning tree for the graph representing communication links between offices as shown above.

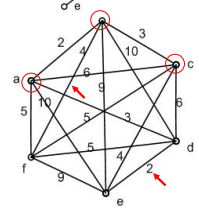
- We now look for the edge which is the cheapest remaining edge or those incident to *a* or *b* or *c* which forms a partial tree. This edge is *ad*.



- Continuing in this manner we find the minimum spanning tree shown:

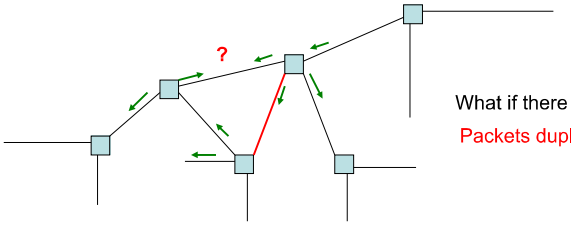


- The total cost of our solution is found to be 2+3+3+2+4=14.



What is the use of minimum spanning tree?

- **City/town planning:** design a minimum-cost road layout connecting several cities
- Used in **communications**: Ethernet **bridge** layout **autoconfiguration** – avoid packets being sent over a network segment twice, so use of minimum spanning tree is required (no cycle)



The shortest path problem

- The weights on a graph may represent *delays* in a communication network or *travel times* along roads.
- A practical problem to consider is to find the **shortest path** between any two vertices.
- **Shortest path → shortest delay**
- The algorithm to determine this will be demonstrated through an example.

Summary

- Definitions of **trees, forests** & **spanning trees**
- Shown **how to draw a spanning tree**
- Introduced the concept of a **minimum spanning tree**
- Presented the **gr$$dy algorithm** for determining a minimum spanning tree: shortest edge first
- Introduced **the shortest path problem**: to find the shortest path between any two vertices in a weighted graph

Readings

- [Mar07] Read 9.5
- [Mar13] Read 9.5

Solution - Stage 1:

- Begin at the start vertex *s*. This is the reference vertex for stage 1.
- Label all the adjacent vertices with the lengths of the paths using only one edge.
- Mark all other vertices with a very large number (larger than the sum of all the weights in the graph). In this case we choose 100. This is shown in the diagram.
- At the same time, start to form a table as shown in Table 1.

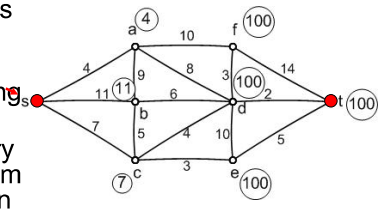


Table 1

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100

- The lengths of paths using only 1 edge from *s*

Introduction to Graph Theory
Lecture 27

Solution - Stage 2:

- Choose as the reference vertex for stage 2 the vertex with the ***smallest label*** that has not already been a reference vertex. This is vertex ***a***.
- Consider any vertex adjacent to the new reference vertex and mark it with the length of the path from *s* via *a* to this vertex if this is less than the current label on the vertex. This gives the labels shown right.
- We also add a new line to Table 1 to give Table 2, noting that as vertex *a* has been made a reference vertex the label of *s* becomes permanent and is marked with an underline in the table.

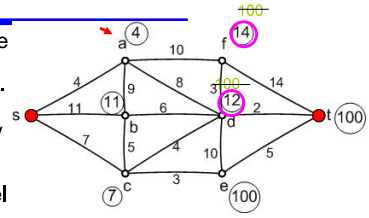


Table 2

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100

- The lengths of paths using up to 2 edges from *s*

Menu

- Shortest path algorithm to determine the shortest path between two vertices of a weighted graph

Solution - Stage 3:

- Choose as the reference vertex the vertex with the ***smallest label*** that has not already been a reference vertex. From table 2 we see that **c** is the reference vertex for stage 3.
- Consider any vertex adjacent to *c* that does not have a permanent label and calculate the length of the path from *s* via *c* to this vertex. If it is less than the current label on the vertex mark the vertex with this length. This gives us the labels shown right.
- We also add a new line to Table 2 to give Table 3. Note that the third line of Table 3 does not have an entry for *a* as this has already been a reference vertex.

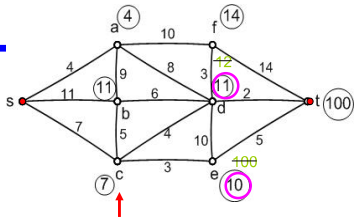


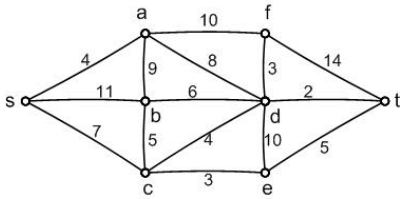
Table 3

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100

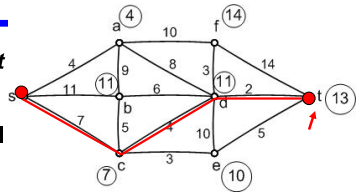
- The lengths of paths using up to 3 edges from *s*

Example 1

- The weighted graph shown below represents a communication network with weights indicating the delays associated with each edge.
- Find the minimum delay path from *s* to *t*.

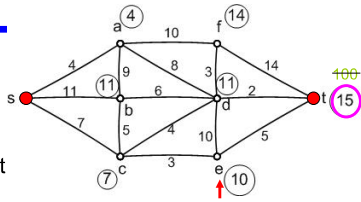


Solution - Stage 7:



- The remaining vertex with the ***smallest label*** is ***t***.
- We therefore give *t* the permanent label of 13.
- As soon as *t* receives a permanent label the algorithm stops as this label is the length of the shortest path from *s* to *t*.
- To find the actual path with this length we ***move backwards*** from *t* looking for *consistent* labels.
- This gives *t d c s*. That is, the path is *s c d t*.

Solution - Stage 4:



- Proceeding as before, the reference vertex for stage 4 is, by inspection of the third line of Table 3, vertex ***e***.
- Again we calculate the lengths of the paths from *s* via *e* to any vertices adjacent to *e* that do not have permanent labels and replace the labels on those vertices with the relevant path lengths if this is less than the existing label.
- This gives the labels shown right and Table 4.

Table 4

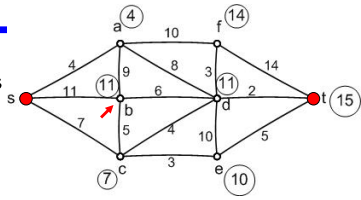
	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	**10**	14	100
e		**11**		**11**		**14**	**15**

- The lengths of paths using up to 4 edges from *s*

Dijkstra's Shortest Path Algorithm (SPA)

- Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.
- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
- For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 9, and the edge connecting it with a neighbor *B* has length 4, then the distance to *B*(through *A*) will be 9 + 4 = 13. If B was previously marked with a distance greater than 13 then change it to 13. Otherwise, keep the current value.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
- If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
- Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Solution - Stage 5:



- Choose ***b*** as the new reference vertex (we could have chosen d instead but this would make no difference to the final result).
- Compare paths from *s* via *b* to the labels on any adjacent vertices with temporary labels and re-label if the paths are found to be shorter.
- The result of stage 5 is that the labels remain as in stage 4, but that the label on b becomes permanent giving Table 5.

Table 5

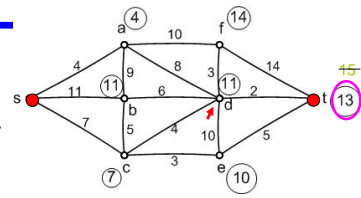
	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100
e		**11**		11		14	15
b				**11**		**14**	**15**

- The lengths of paths using up to 5 edges from *s*

Why is SPA optimal?

- Why SPA gives us the shortest path?
- What is the complexity of SPA?
- Can SPA be generalized for related shortest path problems?

Solution - Stage 6:



- Choose ***d*** as the new reference vertex.
- The only vertices left without permanent labels are now *f* and *t*.
- The path from *s* via *d* to *t* gives *a* smaller value than the current label of 15. Hence we change the label to 11+2=13.
- The new labels are shown right together with Table 6.

Table 6

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100
e		11		11		14	15
b				**11**		14	15
d						**14**	**13**

- The lengths of paths using up to 6 edges from *s*

Summary

- Demonstrated the algorithm to determine the shortest path between two vertices of a weighted graph

Readings

- [Mar07] Read 9.3
- [Mar13] Read 9.3