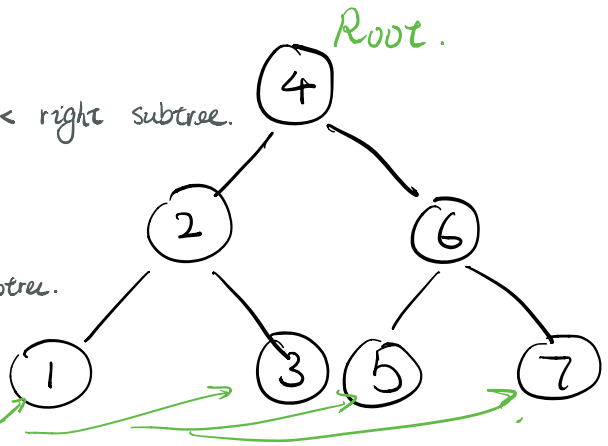
TREEs

Binary Search trees.

1. Ordered:
   left subtree < root < right subtree.
2. Perfectly balanced.
   numbers of nodes
   in left == right subtree.
3. height balanced.
   left == right (height)



--- Level 1

--- Level 2

--- Level 3

❖ Searching in BSTs

Most tree algorithms are best described recursively:

```
TreeContains(tree,key):
|   Input  tree, key
|   Output true if key found in tree, false otherwise
|
|   if tree is empty then
|      return false
|   else if key < data(tree) then
|      return TreeContains(left(tree),key)
|   else if key > data(tree) then
|      return TreeContains(right(tree),key)
|   else          // found
|     return true
|   end if
```

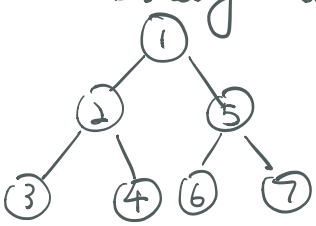
❖ Insertion into BSTs

Insert an item into a tree; item becomes new leaf node

```
TreeInsert(tree,item):
|   Input  tree, item
|   Output tree with item inserted
|
|   if tree is empty then
|      return new node containing item
|   else if item < data(tree) then
|      left(tree) = TreeInsert(left(tree),item)
|      return tree
|   else if item > data(tree) then
|      right(tree) = TreeInsert(right(tree),item)
|      return tree
|   else
|      return tree     // avoid duplicates
|   end if
```

Visit order for binary trees: (Number is order).

1. Preorder (NLR).



Useful for building trees. (Min height).

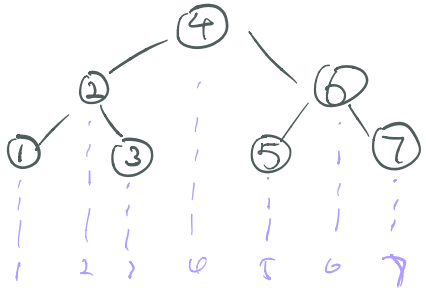
```
showBSTreePreorder(t):
|   Input tree t
|
|   if t is not empty then
|      print data(t)
|      showBSTreePreorder(left(t))
|      showBSTreePreorder(right(t))
|   end if
```

preorder — inorder — post order

↓ the position of print decides.

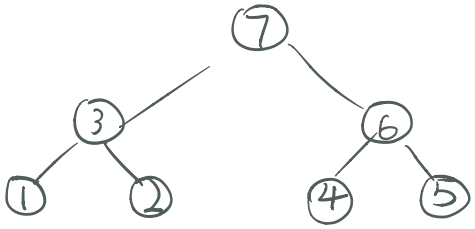
go left all the way till leaf. then back up slowly, and go right once if path exist. Then left.....

2. Inorder (LNR)



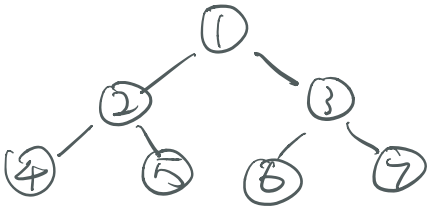
Natural order (from left to right).

3. Postorder (LRN).



Useful for evaluation.

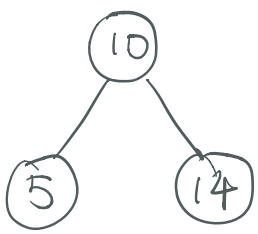
4. Level-order



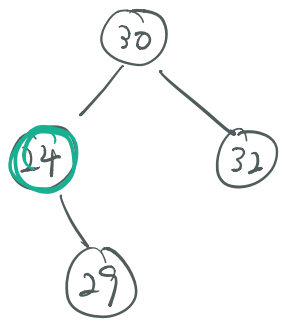
Useful for print(tree).

Tree Join: Useful when the root hasn't been deleted. This method can reduce the depth compared to connecting the next node straight. just rotating and joint.

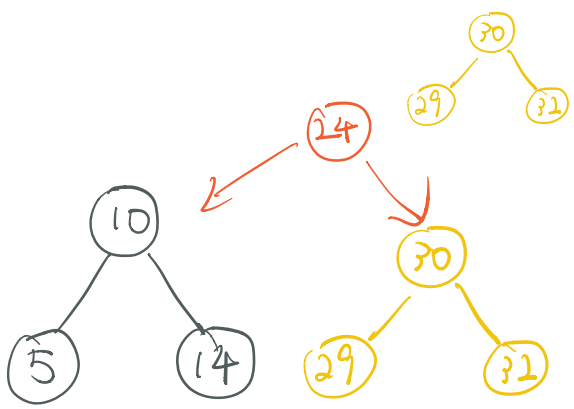
Tree 1

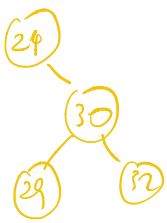


Tree 2



Step 1: Find min node in right subtree.
(OR choose deeper tree, max node if left min node of right)

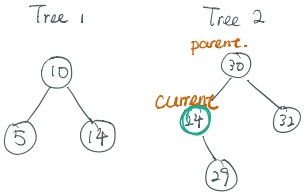


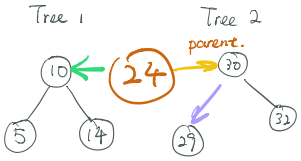


Step 2: Reform T2.
actually rotate right.

Step 3: (24) become new root!

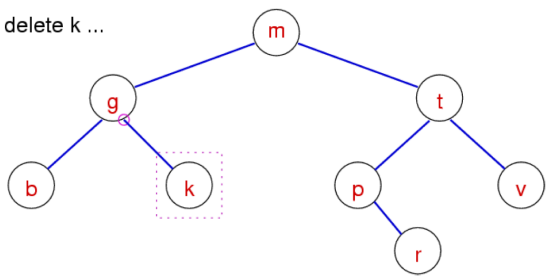
```
TreeJoin(t1,t2):
|  Input  trees t1,t2
|  Output t1 and t2 joined together
|
|  if t1 is empty then return t2
|  else if t2 is empty then return t1
|  else
|  |  curr=t2, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t2
|  |     parent=curr
|  |     curr=left(curr)
|  |  end while
|  |  if parent≠NULL then
|  |     left(parent)=right(curr)  // unlink min element from parent
|  |     right(curr)=t2
|  |  end if
|  |  left(curr)=t1
|  |  return curr                  // curr is new root
|  end if
```





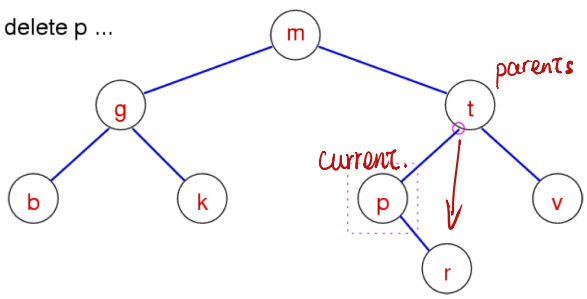
Delete element from BST.

Case 2: item to be deleted is a leaf (zero subtrees)



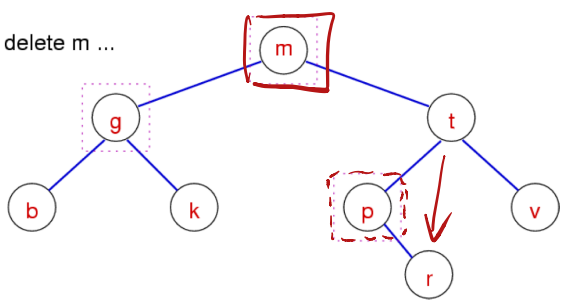
Just delete the item

Case 3: item to be deleted has one subtree



Replace the item by its only subtree

Case 4: item to be deleted has two subtrees



Use join tree method.

Find P,
Set it as new root ....

```
TreeDelete(t,item):
|  Input  tree t, item
|  Output t with item deleted
|
|  if t is not empty then          // nothing to do if tree is empty
|  |  if item < data(t) then       // delete item in left subtree
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if item > data(t) then  // delete item in left subtree
|  |     right(t)=TreeDelete(right(t),item)
|  |  else                         // node 't' must be deleted
|  |  |  if left(t) and right(t) are empty then
|  |  |     new=empty tree                    // 0 children
|  |  |  else if left(t) is empty then
|  |  |     new=right(t)                      // 1 child
|  |  |  else if right(t) is empty then
|  |  |     new=left(t)                       // 1 child
|  |  |  else
|  |  |     new=TreeJoin(left(t),right(t))    // 2 children
|  |  |  end if
|  |  |  free memory allocated for t
|  |  |  t=new
|  |  end if
|  end if
|  return t
```

Searching items.

formed the node.

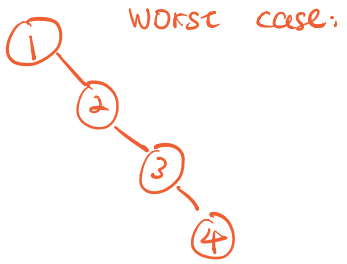
case 2: leaf node/root

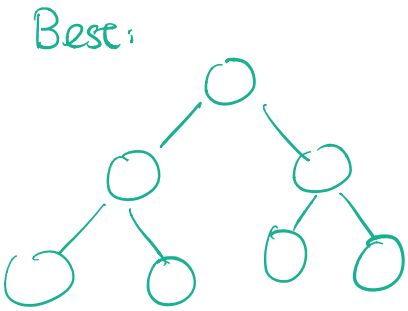
what does new Tree do

?

Balancing Binary Search Trees

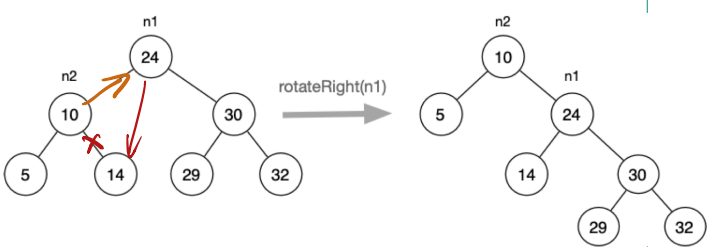
- worst case: keys inserted in ascending/descending order
  (effectively have a linked list, so search cost is *O(n)*)
- best case (for at-leaf insertion): keys inserted in pre-order
  (tree height ⇒ search cost is *O(log n)*; tree is balanced)
- average case: keys inserted in random order
  (tree height ⇒ search cost is *O(log n)*; but cost ≥ best case)





Tree Rotation: { low cost : O(1)
Improve tree balance.

Example of right rotation:



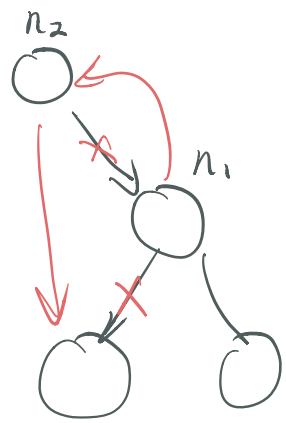
More left child to root, rearrange links to retain other.

Algorithm for right rotation:

```
rotateRight(n1):
|   Input  tree n1
|   Output n1 rotated to the right
|
|   if n1 is empty ∨ left(n1) is empty then
|      return n1
|   end if
|   n2=left(n1)
|   left(n1)=right(n2)
|   right(n2)=n1
|   return n2
```

Algorithm for left rotation:

```
rotateLeft(n2):
|   Input  tree n2
|   Output n2 rotated to the left
|
|   if n2 is empty ∨ right(n2) is empty then
|      return n2
|   end if
|   n1=right(n2)
|   right(n2)=left(n1)
|   left(n1)=n2
|   return n1
```

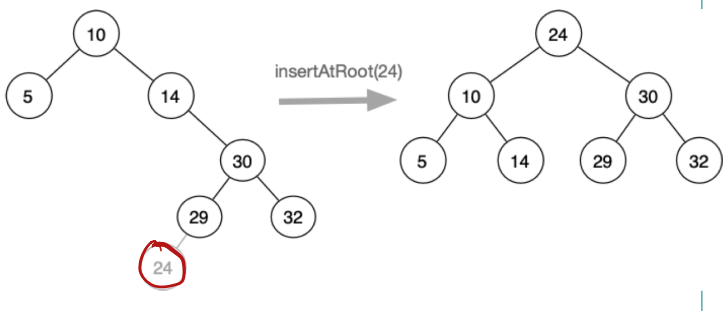


Inserting ~~at~~ into root

- complex when insert ✗
- More likely to be search, as this is recently added item. ✓
  - Lower cost.

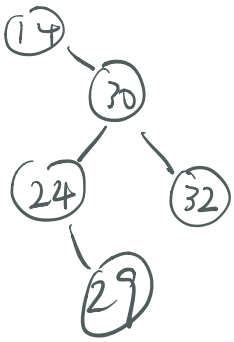
Example of inserting at root:

Implemented by rotation one level at a time.

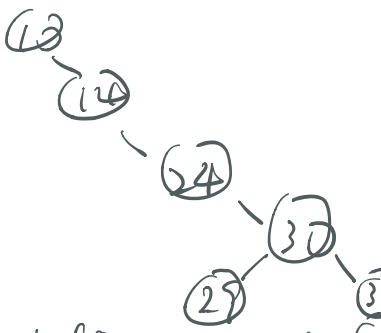


Process:

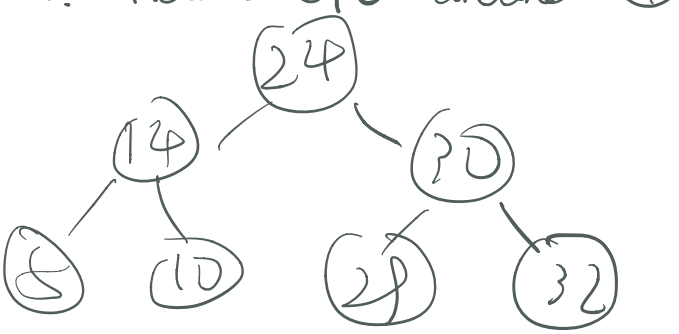
1). Right rotation around 29



2). Rotate right around 30



3). Rotate left around 14



Algorithm for inserting at root:

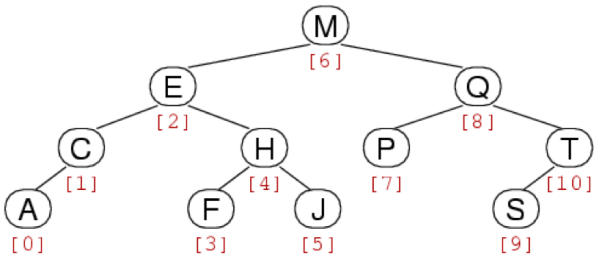
```
insertAtRoot(t, it):
|   Input tree t, item it to be inserted
|   Output modified tree with item at root
|
|   if t is empty tree then
|      t = new node containing item
|   else if item < root(t) then
|      left(t) = insertAtRoot(left(t), it)
|      t = rotateRight(t)
|   else if it > root(t) then
|      right(t) = insertAtRoot(right(t), it
|      t = rotateLeft(t)
|   end if
|   return t;
```

① ② will go to the appropriate inserted location first, then rotate left if insert right, and vice versa.

Tree Partition(tree, i).

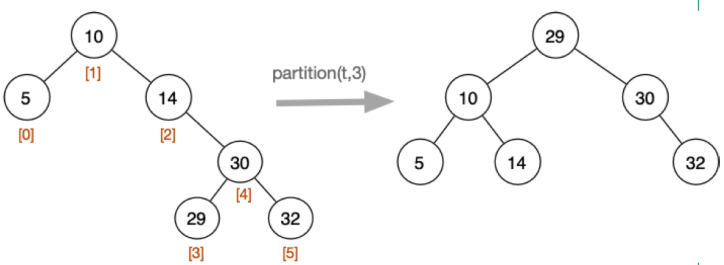
Choose suitable index in the middle., to improve balance.

- Re-arrange the tree so that index i became the root.
  (Promote index i to root)



For tree with *N* nodes, indices are *0* .. *N-1*, in LNR order

Example of partition:




```
partition(tree,i):
|   Input  tree with n nodes, index i
|   Output tree with i^th item moved to the root
|
|   m=#nodes(left(tree))
|   if i < m then
|     left(tree)=partition(left(tree),i)
|     tree=rotateRight(tree)
|   else if i > m then
|     right(tree)=partition(right(tree),i-m-1)
|     tree=rotateLeft(tree)
|   end if
|   return tree
```

check the nums of node in left. To see the location of index.

① in left subtree or right subtree.

Note: size(tree) = n, size(left(tree)) = m, size(right(tree)) = n-m-1

①② are searching the position of index i.

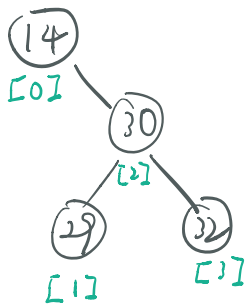
③ At this time, we exclude the left subtree, so the index must update, e.g.

As in the example,

index = 3 , m = 1

i - m - 1 = 3 - 1 - 1 = 1

Total, left subtree, root



The index must update, since we changed the root, but the wanted node is 29 at the time, regardless its index.

Periodic Rebalancing.

Move median item to the root, every after k insertions.

- insert at leaves as before; periodically, rebalance the tree

```
Input  tree, item
Output tree with item randomly inserted

t=insertAtLeaf(tree,item)
if #nodes(t) mod k = 0 then
   t=rebalance(t)
end if
return t
```

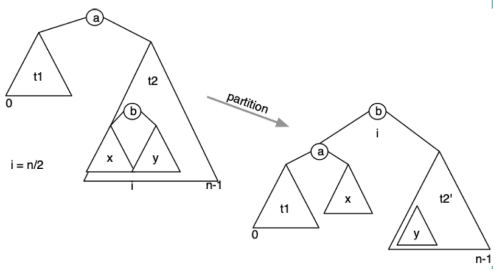
When to rebalance? e.g. after every *k* insertions

Implementation of rebalance:

```
rebalance(t):
  Input  tree t with n nodes
  Output t rebalanced

  if n≥3 then
    // put node with median key at root
    t=partition(t,⌊n/2⌋)
    // then rebalance each subtree
    left(t)=rebalance(left(t))
    right(t)=rebalance(right(t))
  end if
  return t
```

Last page

How to rebalance a BST? Move median item to root.



Randomise BST insertion: Likely to improve structure / more balanced trees.

Complexity: O(log n)

Approach: normally do leaf insert, randomly do root insert.

```
insertRandom(tree,item)
  Input  tree, item
  Output tree with item randomly inserted

  if tree is empty then
     return new node containing item
  end if
  // p/q chance of doing root insert
  if random() mod q < p then
     return insertAtRoot(tree,item)
  else
     return insertAtLeaf(tree,item)
  end if
```

① Set q/p = 0.3 = 30%
if random generated a number < 30% do
...
else do
...

E.g. 30% chance ⇒ choose *p=3, q=10*

Splay tree:

(= double rotating version of Inserting at root).

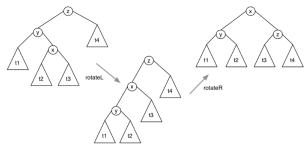
Complexity:
$O_{Ave}((n+m)\log_2(n+m))$
$O_{worst}(n)$
m (insert + search) operations,
n nodes,

- A self-balancing tree.
  - ↳ more recent search item will be rotated close to root.
  - ↳ Improving structure

- Difference compare to inserting at root:
  1). Splay tree considers 3 levels of trees ← parent, child, Grandchild.
  2). Performing double-rotating for three levels analysis.

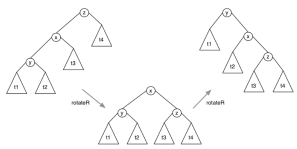
1. Most of the time, double-rotation can improve the structure more effective then single rotation.

Example: double-rotation case for right-child of left-child:



2. Exceptions: ....

Example: double-rotation case for left-child of left-child:



Question: Compare to Periodic balance.
Splay tree is the regular version of periodic balance?

```
insertSplay(tree,item):
|   Input  tree, item
|   Output tree with item splay-inserted
|
|   if tree is empty then return new node containing item
|   else if item=data(tree) then return tree
|   else if item < data(tree) then
|   |   if left(tree) is empty then
|   |      left(tree) = new node containing item
|   |   else if item < data(left(tree)) then
|   |         // Case 1: left-child of left-child
|   |      tll = insertSplay(tll),item)
|   |   (3) tree = rotateRight(tree)
|   |   else  // Case 2: right-child of left-child
|   |      tlr = insertSplay(tlr,item)
|   |   (5) left(tree) = rotateLeft(left(tree))
|   |   end if
|   |   return (1) rotateRight(tree)
|   else if item > data(tree) then
|   |   if right(tree) is empty then
|   |      right(tree) = new node containing item
|   |   else if item < data(right(tree)) then
|   |         // Case 3: left-child of right-child
|   |      trl = insertSplay(trl,item)
|   |      right(tree) = (6) rotateRight(right(tree))
|   |   else  // Case 4: right-child of right-child
|   |      trr = insertSplay(trr,item)
|   |   (4) tree = rotateLeft(tree)
|   |   end if
|   |   return rotateLeft(tree)
|   end if  (2)
```

Answer: No!

- Periodic balance is promoting a random middle number to the root, then rebalance each tree.

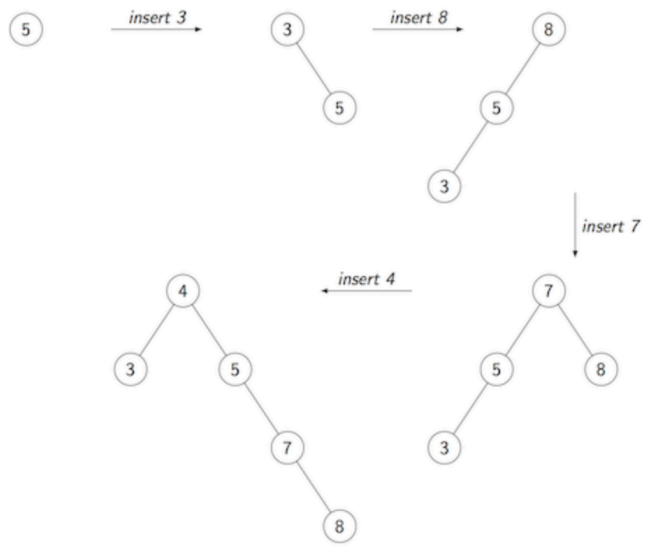
Basically,
Periodic balance: (Partition + Rebalance) < left / right — every certain period

- Splay tree is (searching) inserting + double rotating the required number onto the root.

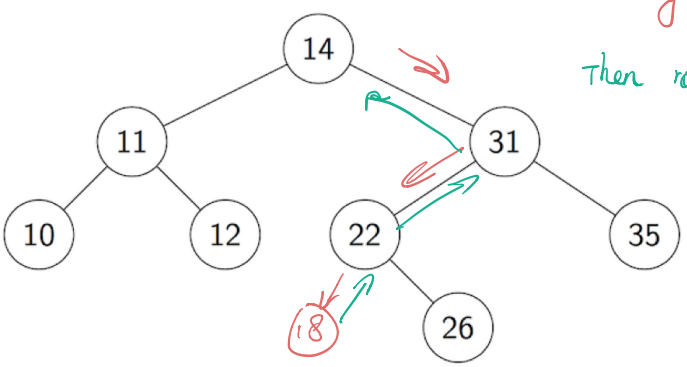
①② First level rotation, rotate right if in left, vice versa.
③④ Second level rotation. rotate root tree twice if tll vice versa.
⑤⑥ Second level rotation, rotate next level tree.

Insert 5, 3, 8, 7, 4 into empty tree.

This is how the tree would grow if root-root rotation were used for the left-left-child and right-right-child cases, as in the standard implementation of splay trees:



Example: insert **18** into this splay tree:



Because 18 > 14

Inserting in right subtree.

Then rotate right → right → left.

Moving desired item into root:

Searching in splay trees:

```
searchSplay(tree,item):
|  Input  tree, item
|  Output address of item if found in tree
|         NULL otherwise
|
|  if tree=NULL then
|     return NULL
|  else
|  |  tree = splay(tree,item)
|  |  if data(tree)=item then
|  |     return tree
|  |  else
|  |     return NULL
|  |  end if
|  end if
```

Move to root.

then return tree.

splay() is similar to **insertSplay()**, but doesn't add a node
moves **item** to root if found, moves nearest node to root if not found

All the operation above

1. Randomised inserting.
2. Occasionaly rebalancing
   - Periodic rebalancing
   - partitioning
3. Splay tree.

All have worse case O(n).

BREAK.

Ideally, we want both average/worst case to be *O(log n)*

- AVL trees … fix imbalances as soon as they occur
- 2-3-4 trees … use varying-sized nodes to assist balance
- red-black trees … isomorphic to 2-3-4, but binary nodes

AVL Tree:

√: · Well balanced O(log2n)

· Cheap to fixing imbalance.

↳ fixing locally. would not affect whole structure.

A tree is unbalanced when abs(height(left)-height(right)) > 1

This can be repaired by rotation:

- if left subtree too deep, rotate right
- if right subtree too deep, rotate left

Problem: determining height/depth of subtrees is expensive

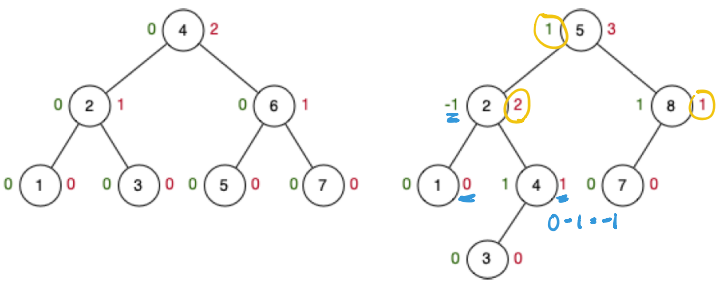
- need to traverse whole subtree to find longest path

Solution: store balance data in each node (either height or balance)

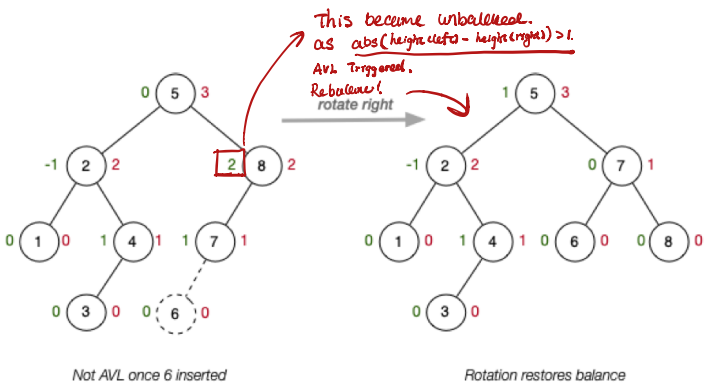
- but extra effort needed to maintain this data on insertion

Red numbers are height; green numbers are balance

② - ① = ①
height. difference.



How an unbalanced tree can be rebalanced



Implementation of AVL insertion

```
insertAVL(tree,item):
|   Input  tree, item
|   Output tree with item AVL-inserted
|
|   if tree is empty then
|      return new node containing item
|   else if item = data(tree) then
|      return tree
|   else
|   |   if item < data(tree) then
|   |      left(tree) = insertAVL(left(tree),item)
|   |   else if item > data(tree) then
|   |      right(tree) = insertAVL(right(tree),item)
|   |   end if
|   |   LHeight = height(left(tree))
|   |   RHeight = height(right(tree))
|   |   if (LHeight - RHeight) > 1 then
|   |      if item > data(left(tree)) then
|   |         left(tree) = rotateLeft(left(tree))
|   |      end if
|   |      tree=rotateRight(tree)
|   |   else if (RHeight - LHeight) > 1 then
|   |      if item < data(right(tree)) then
|   |         right(tree) = rotateRight(right(tree))
|   |      end if
|   |      tree=rotateLeft(tree)
|   |   end if
|   |   return tree
|   end if
```

we return true then insert.

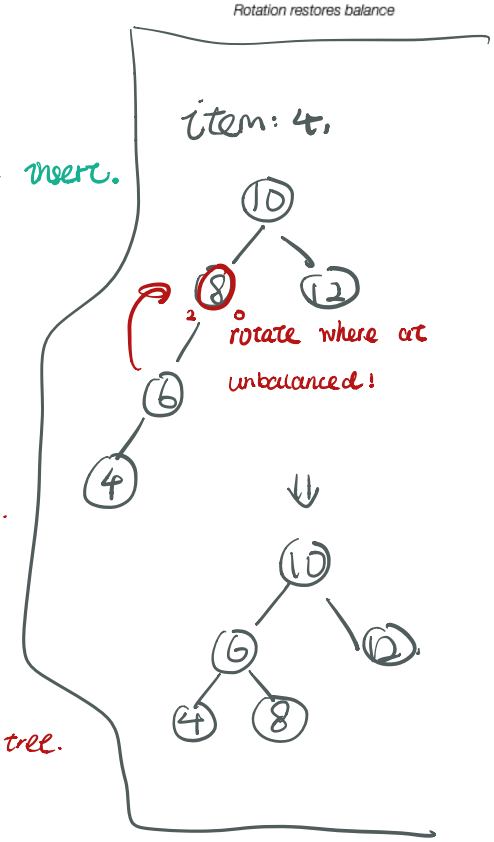
we return false

insert.

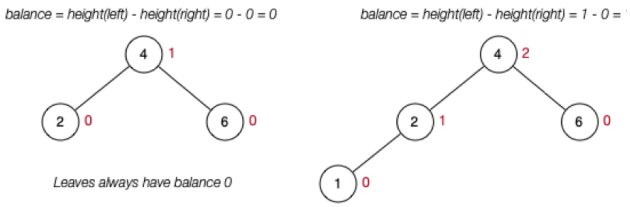
check balance.

→ check one more level.

→ Return balancing tree.



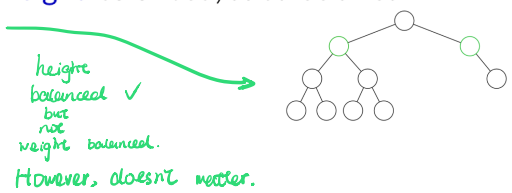
Store height in nodes; update on insertion; compute balance



If abs(balance) > 1 after updating, rebalance via rotation(s)

Analysis of AVL trees:

- trees are height-balanced; subtree depths differ by +/-1
- average/worst-case search performance of *O(log n)*
- *require* extra data to be stored in each node (efficiency)
- *require* extra data to be maintained during insertion
- may not be weight-balanced; subtree sizes may differ



2-3-4 Trees.

Binary tree will normally have search cost of $O(\log_2 n)$.

2 means 2 branches and n based on the depth of trees.

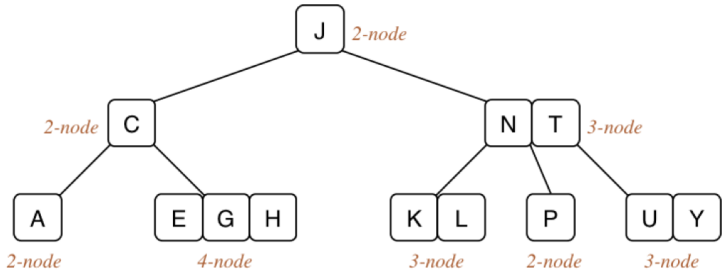
However, what if $\log_2 n$, 2 is 4 or 2 vs 8?

$\log_2 4096 = 12$
$\log_4 4096 = 6$
$\log_8 4096 = 4$

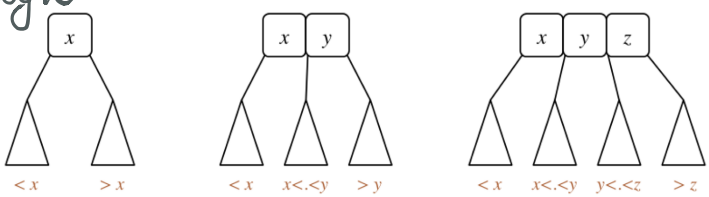
↑ Search cost. Max (searching path).

2-3-4 trees have three kinds of nodes

- 2-nodes, with two children (same as normal BSTs)
- 3-nodes, two values and three children
- 4-nodes, three values and four children



Same logic:



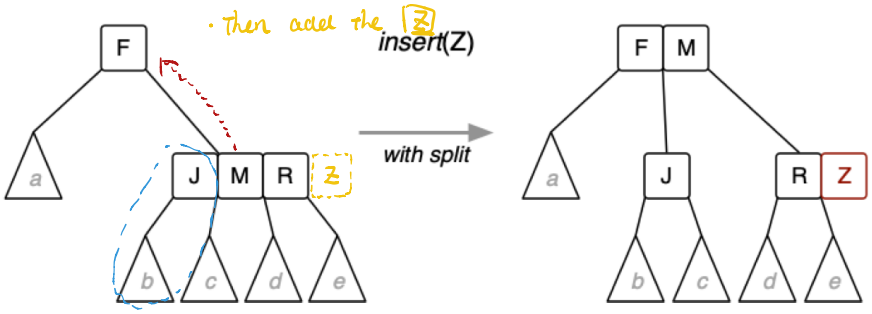
But:
2-3-4 tree grows upward via node splitting.

Node splitting:

Insertion into a full node causes a split

- middle value propagated to parent node
- values in original node split across original node and new node

. Promote the middle node into parents' node, as this can't fit in.
. Now parent has 2 nodes F M ⇒ should have 3 children.
. Then added the Z

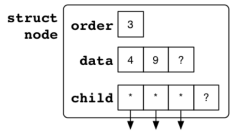


Searching in 2-3-4 trees:

```
Search(tree,item):
|  Input  tree, item
|  Output address of item if found in 2-3-4 tree
|         NULL otherwise
|
|  if tree is empty then
|     return NULL
|  else
|  |  scan tree.data to find i such that
|  |     tree.data[i-1] < item ≤ tree.data[i]
|  |  if item=tree.data[i] then    // item found
|  |     return address of tree.data[i]
|  |  else        // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

Possible concrete 2-3-4 tree data structure:

```
typedef struct node {
   int          order;      // 2, 3 or 4
   int          data[3];    // items in node
   struct node *child[4];   // links to subtrees
} node;
```



Only this part is different to regular binary search.

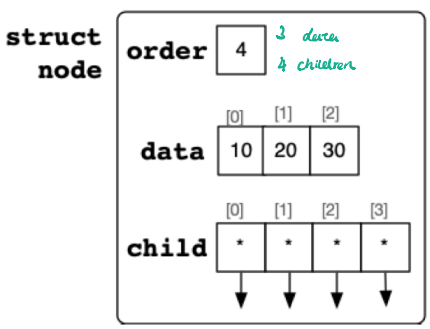
Before: Left is smaller than node etc.
Now: we need to scan the node, to see what the values are.

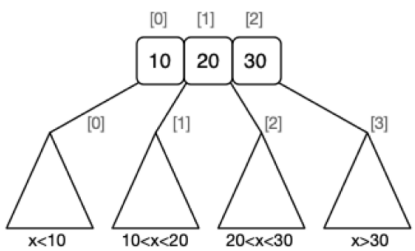
Example:

Finding which branch to follow

```
// n is a pointer to a (struct node)
int i;
for (i = 0; i < n->order-1; i++) {
   if (item <= n->data[i]) break;
}
// go to the i^th subtree, unless item == n->data[i]
```

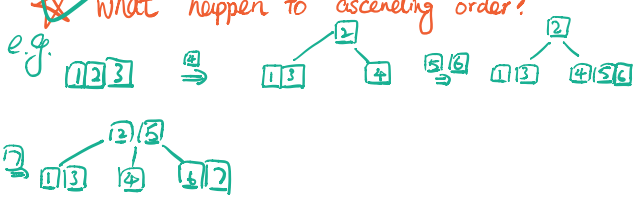
desired value.





Complexity: { Worst: $O(\log_2 n)$; Best: $O(\log_4 n)$

what happen to ascending order?



Insertion in 2-3-4 tree. (& Inserting at leaf)

Insertion algorithm:

- find leaf node where Item belongs (via search)
- if not full (i.e. order < 4)
  - insert Item in this node, order++
- if node is full (i.e. contains 3 items)
  - split into two 2-nodes as leaves
  - promote middle element to parent — We need to keep track of parent.
  - insert item into appropriate leaf 2-node
  - if parent is a 4-node
    - continue split/promote upwards
  - if promote to root, and root is a 4-node
    - split root node and add new root

Insertion algorithm:

```
insert(tree,item):
|  Input  2-3-4 tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|  node=Search(tree,item)
|  parent=parent of node
|  if node.order < 4 then
|     insert item into node
|     increment node.order
|  else
|  |  promote = node.data[1]     // middle value promoting.
|  |  nodeL   = new node containing data[0]
|  |  nodeR   = new node containing data[2]
|  |  delete node
|  |  if item < promote then
|  |     insert(nodeL,item)
|  |  else                          } then insert.
|  |     insert(nodeR,item)
|  |  end if
|  |  insert(parent,promote)
|  |  while parent.order=4 do
|  |     continue promote/split upwards
|  |  end while
|  |  if parent is root ∧ parent.order=4 then
|  |     split root, making new root
|  |  end if
|  end if
```

Variations on 2-3-4 trees ...

Variation #1: why stop at 4? why not 2-3-4-5 trees? or *M*-way trees?

- allow nodes to hold between *M/2* and *M-1* items $[\frac{M}{2} \text{ to } M-1]$ half full
- if each node is a disk-page, then we have a B-tree (databases) $\log_{400} n$
- for B-trees, depending on **`Item`** size, *M > 100/200/400*

Variation #2: don't have "variable-sized" nodes

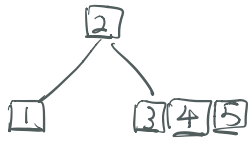
- use standard BST nodes, augmented with one extra piece of data
- implement similar strategy as 2-3-4 trees → red-black trees.

Example :

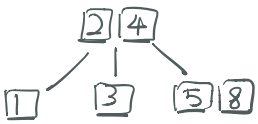
insert 1 2 3 4 5 8 6 7 9 10

4:

5:



8 :

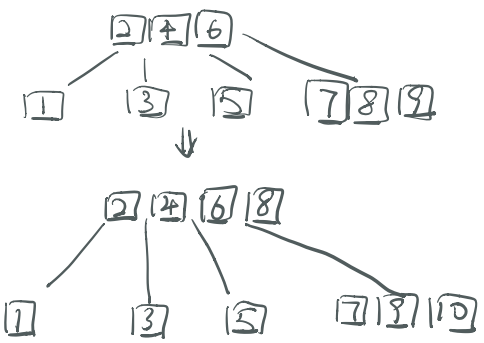


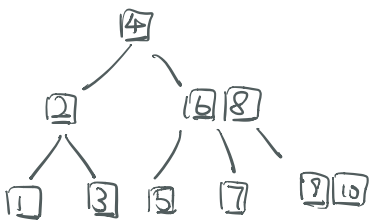
6 :

7:

9:

10:

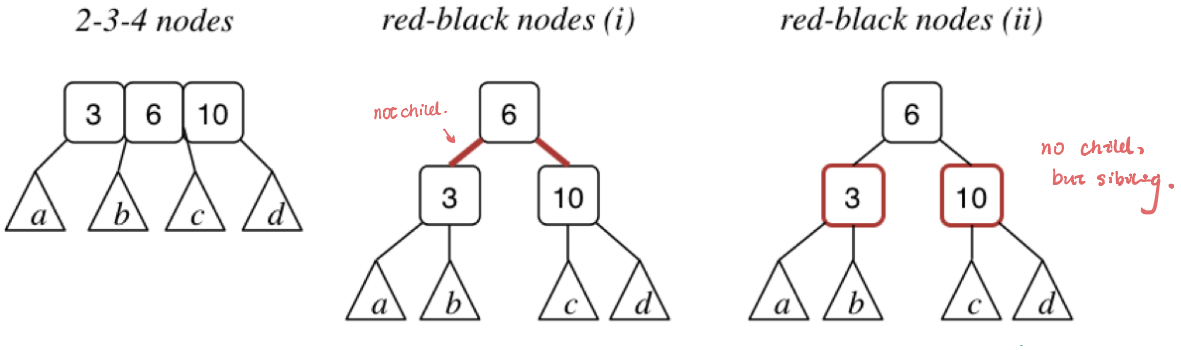


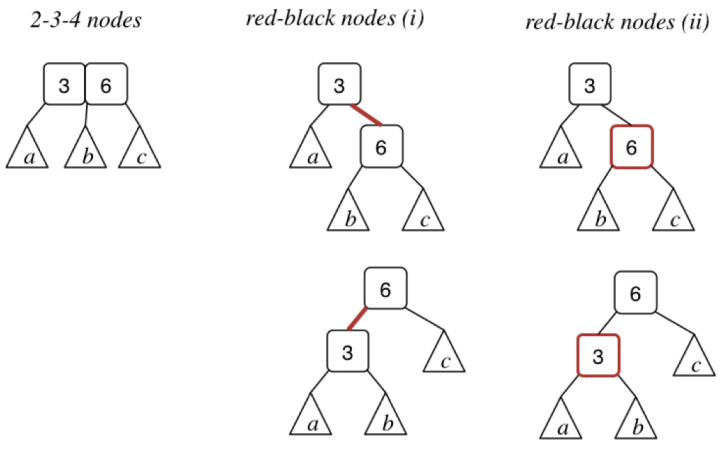


Red-Black Tree (a 2-3-4 tree with only two nodes).

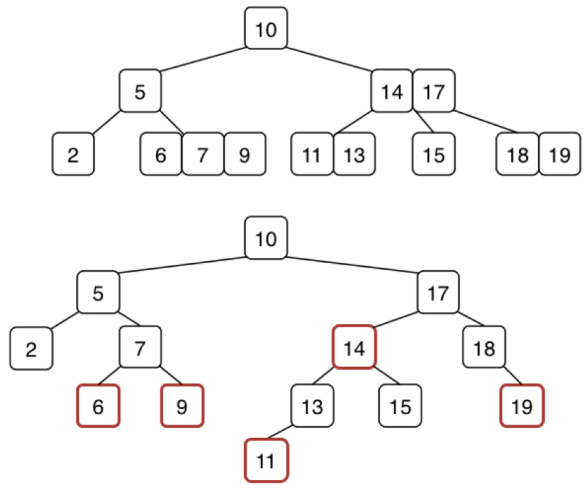
- Red ⇒ combine the nodes to represent -3 or -4 nodes.
- Black ⇒ ordinary BST lines.

Advantage:
- BST way of searching.
- Auto balance like 2-3-4 tree.





Equivalent trees (one 2-3-4, one red-black):



Red-black tree implementation:

```
typedef enum {RED,BLACK} Colour;
typedef struct node *RBTree;
typedef struct node {
   int    data;    // actual data
   Colour colour;  // relationship to parent
   RBTree left;    // left subtree
   RBTree right;   // right subtree
} node;

#define colour(tree) ((tree) != NULL && (tree)->colour)
#define isRed(tree)  ((tree) != NULL && (tree)->colour == RED)
```

RED = node is part of the same 2-3-4 node as its parent (sibling)

BLACK = node is a child of the 2-3-4 node containing the parent

New nodes are always red ...

```
RBTree newNode(Item it) {
    RBTree new = malloc(sizeof(Node));
    assert(new != NULL);
    data(new) = it;
    color(new) = RED;
    left(new) = right(new) = NULL;
    return new;
}
```

.. because they're always inserted into a leaf node

Searching method is same as BST.

However, insertion is more complicated.

Insertion is more complex than for standard BSTs

- need to recall direction of last branch (L or R)
- need to recall whether parent link is red or black
- splitting/promoting implemented by rotateLeft/rotateRight

Several cases to consider depending on colour/direction combinations

High-level description of insertion algorithm:

```
insertRedBlack(tree,item):
|  Input  red-black tree, item
|  Output tree with item inserted
|
|  tree = insertRB(tree,item,false)
|  colour(tree) = BLACK
|  return tree
```

indicate whether the tree is located in the left or right, false, true.

This function acts as a "wrapper" around the recursive function. Cannot call recursive straightaway.

Having restructured the tree, it then makes the root node BLACK

Overview of the recursive function ...

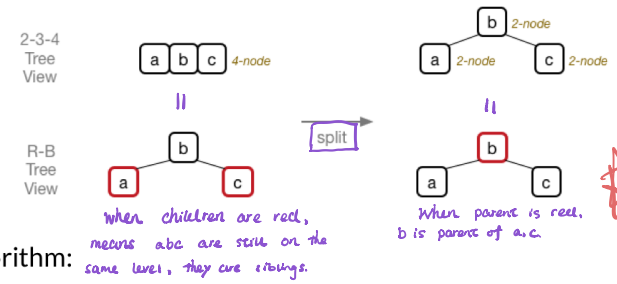
```
insertRB(tree,item,inRight):
|   Input  tree, item, inRight
|          // inRight = direction of last branch
|   Output tree with item inserted
|
|   if tree is empty then return newNode(item)
|   if data(tree) = item then return tree
|   if isRed(left(tree)) ∧ isRed(right(tree)) then
|      split 4-node
|   end if
|   recursive insert cases (cf. regular BST)
|   re-arrange links/colours after insert
|   return modified tree
```

No duplicated value, return if found

and

four node.

Splitting a 4-node, in a red-black tree:



when children are red, means abc are still on the same level, they are siblings.

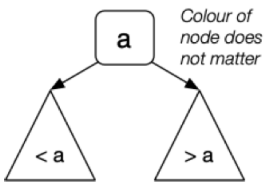
When parent is red, b is parent of a, c.

Why just change the colour?

Algorithm:

```
if isRed(left(tree)) ∧ isRed(right(tree)) then
   colour(tree) = RED
   colour(left(tree)) = BLACK
   colour(right(tree)) = BLACK
end if
```

Simple recursive insert (a la BST):



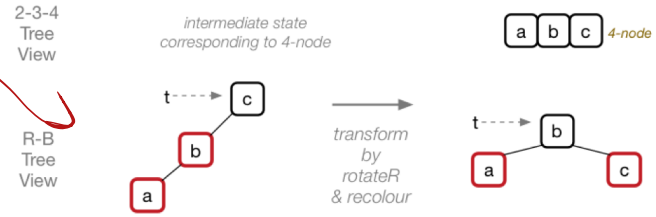
Algorithm:

```
if item < data(tree)) then
   left(tree) = insertRB(left(tree),item,false)
   re-arrange links/colours after insert
else        // item larger than data in root
   right(tree) = insertRB(right(tree),item,true)
   re-arrange links/colours after insert
end if
```

left

right.

Re-arrangement #1: two successive red links = newly-created 4-node



Algorithm:

```
if isRed(left(tree)) ∧ isRed(left(left(tree))) then
   tree=rotateRight(tree)
   colour(tree)=BLACK
   colour(right(tree))=RED
end if
```

Re-arrangement #2: "normalise" direction of successive red links

Algorithm:

```
if inRight ∧ isRed(tree) ∧ isRed(left(tree)) then
   tree=rotateRight(tree)
end if
```

Cost analysis for red-black trees:

- tree is well-balanced; worst case search is $O(\log_2 n)$
- insertion affects nodes down one path; max #rotations is $2 \cdot h$
  (where h is the height of the tree)

Only disadvantage is complexity of insertion/deletion code.

Note: red-black trees were popularised by Sedgewick.

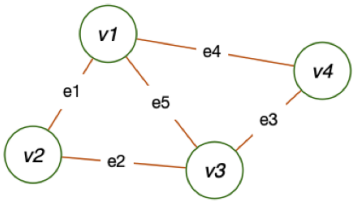
Graphs.

Graph Basics.

A graph $G = (V,E)$

- V is a set of vertices
- E is a set of edges (subset of $V \times V$)

Example:



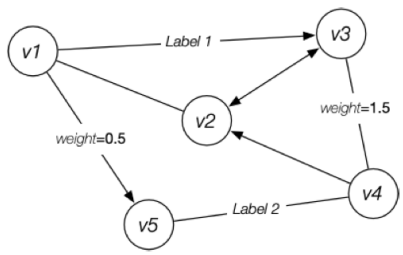
V = { v1, v2, v3, v4 }

E = { e1, e2, e3, e4, e5 }
or
E = { (v1,v2), (v2,v3),
(v3,v4), (v1,v4), (v1,v3) }

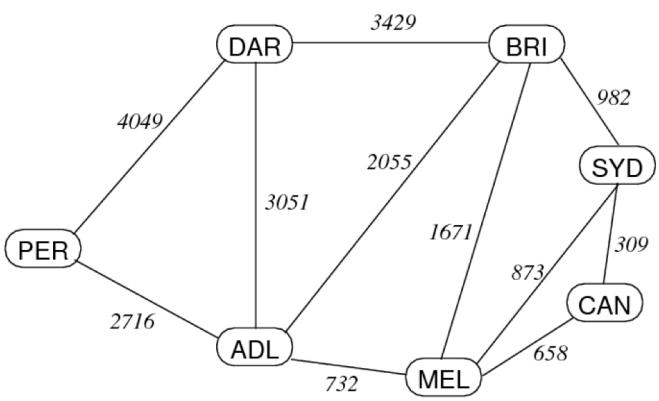
weights and direction:

Nodes are distinguished by a unique identifier

Edges may be (optionally) directed, labelled and/or weighted



Map examples:



A real example: Australian road distances

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

Questions we might ask about a graph:

- is there a way to get from item A to item B?
- what is the best way to get from A to B?
- which items are directly connected (A↔B)?

Graph algorithms are generally more complex than tree/list ones:

- no implicit order of items
- graphs may contain cycles
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

Properties:

Sparse and dense dovegram:

Terminology: $|V|$ and $|E|$ (cardinality) normally written just as V and E.

A graph with V vertices has at most $V(V-1)/2$ edges.

The ratio $E:V$ can vary considerably.

More edges (links), less vertices (nodes) → dense.

- if E is closer to V^2, the graph is dense
- if E is closer to V, the graph is sparse
  - Example: web pages and hyperlinks

Knowing whether a graph is sparse or dense is important

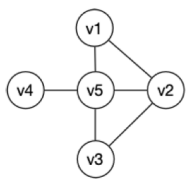
- may affect choice of data structures to represent graph
- may affect choice of algorithms to process graph

For an edge e that connects vertices v and w

- v and w are adjacent (neighbours)
- e is incident on both v and w

Degree of a vertex v

- number of edges incident on e



degree(v1) = 2
degree(v2) = 3
degree(v3) = 2
degree(v4) = 1
degree(v5) = 4

Degree ⇓ the number of ways to connect with outside world.

Synonyms:

- vertex = node
- edge = arc = link (Note: some people use arc for *directed* edges)

Path and cycle:

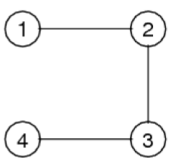
Path: a sequence of vertices where

- each vertex has an edge to its predecessor

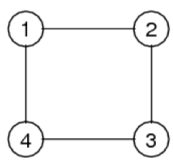
Cycle: a path where

- last vertex in path is same as first vertex in path

Length of path or cycle = #edges



Path: 1–2, 2–3, 3–4



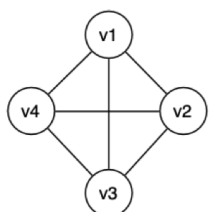
Cycle: 1–2, 2–3, 3–4, 4–1

Connected graph

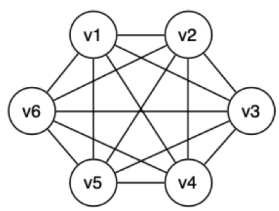
- there is a *path* from each vertex to every other vertex
- if a graph is not connected, it has ≥2 connected components

Complete graph K_V

- there is an *edge* from each vertex to every other vertex
- in a complete graph, $E = V(V-1)/2$



Complete Graphs

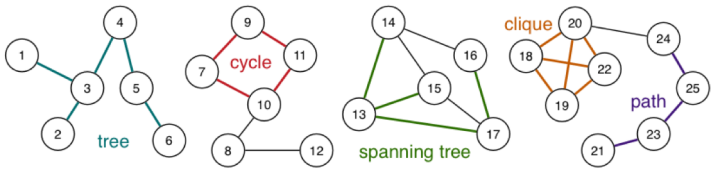


Tree: connected (sub)graph with no cycles

Spanning tree: tree containing all vertices

Clique: complete subgraph

Consider the following *single graph*:



This graph has 25 vertices, 32 edges, and 4 connected components

Note: The entire graph has no spanning tree; what is shown in green is a spanning tree of the third connected component

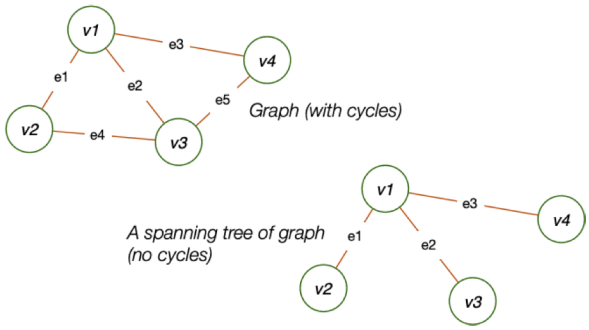
A spanning tree of connected graph *G = (V,E)*

- is a subgraph of *G* containing all of *V*
- and is a single tree (connected, no cycles)

A spanning forest of non-connected graph *G = (V,E)*

- is a subgraph of *G* containing all of *V*
- and is a set of trees (not connected, no cycles),
  - with one tree for each *connected component*

Can form spanning tree from graph by removing edges



Many possible spanning trees can be formed. Which is "best"?

Directed and Undirected

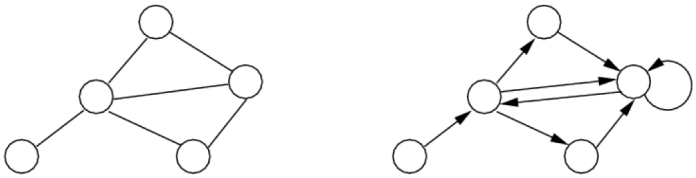
Undirected graph

- *edge(u,v) = edge(v,u)*, no self-loops (i.e. no *edge(v,v)*)

Directed graph

- *edge(u,v) ≠ edge(v,u)*, can have self-loops (i.e. *edge(v,v)*)

Examples:



Undirected graph *Directed graph*

Graph Representation.

1. Array of edges.

Cost

Storage cost: *O(E)*

Cost of operations:

- initialisation: *O(1)*
- insert edge: *O(E)* (need to check for edge in array)
- delete edge: *O(E)* (need to find edge in edge array)

If array is full on insert

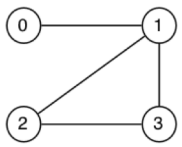
- allocate space for a bigger array, copy edges across ⇒ still *O(E)*

If we maintain edges in order

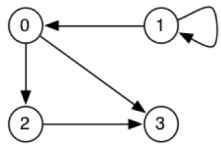
- use binary search to find edge ⇒ *O(log E)*

Edges are represented as an array of **`Edge`** values (= pairs of vertices)

- space efficient representation
- adding and deleting edges is slightly complex
- undirected: order of vertices in an **`Edge`** doesn't matter
- directed: order of vertices in an **`Edge`** encodes direction



[(0,1), (1,2), (1,3), (2,3)]

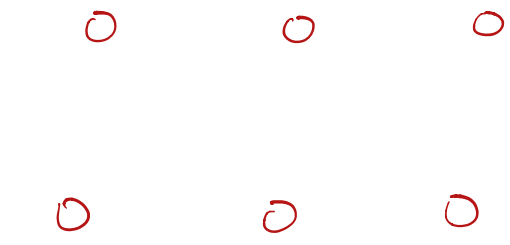


[(1,0), (1,1), (0.2), (0,3), (2,3)]

For simplicity, we always assume vertices to be numbered **`0..V-1`**

Graph initialisation

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph (no edges)
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate enough memory for g.edges[]
|   return g
```



Assumes ≅ **`struct Graph { int nV; int nE; Edge edges[]; }`**

Number of vertices

Number of edges

No edge yet.

Edge insertion

```
insertEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g containing (v,w)
|
|   i=0
|   while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|      i=i+1
|   end while
|   if i=g.nE then       // (v,w) not found
|      g.edges[i]=(v,w)
|      g.nE=g.nE+1
|   end if
```

check the existence of edges, do nothing if found, else insert.

We "normalise" edges so that e.g (v < w) in all (v,w)

Edge removal

```
removeEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g without (v,w)
|
|   i=0
|   while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|      i=i+1
|   end while
|   if i < g.nE then  // (v,w) found
|      g.edges[i]=g.edges[g.nE-1]
|           // replace by last edge in array
|      g.nE=g.nE-1
|   end if
```

Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nE-1 do
|   |   (v,w)=g.edges[i]
|   |   print v"—"w
|   end for
```

2. Adjacency Representation

Cost →

Storage cost: $O(V^2)$

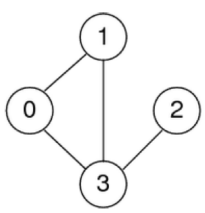
If the graph is sparse, most storage is wasted.

Cost of operations:

- initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
- insert edge: $O(1)$ (set two cells in matrix)
- delete edge: $O(1)$ (unset two cells in matrix)

Edges represented by a $V \times V$ matrix

if 0 and 1 have connection, then shows "1", else "0"

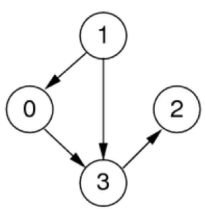


Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0

content of vertices

Similarly, if A can go to B, then appear as "1".



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

Advantages

- easily implemented as 2-dimensional array
- can represent graphs, digraphs and weighted graphs
  - graphs: symmetric boolean matrix
  - digraphs: non-symmetric boolean matrix
  - weighted: non-symmetric matrix of weight values

Disadvantages:

- if few edges (sparse) ⇒ memory-inefficient ($O(V^2)$ space)

Graph initialisation

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[][]
|   for all i,j=0..V-1 do
|      g.edges[i][j]=0    // false
|   end for
|   return g
```

New graph with no edges.

Edge insertion

```
insertEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g containing (v,w)
|
|   if g.edges[v][w] = 0 then  // (v,w) not in graph
|      g.edges[v][w]=1     // set to true
|      g.edges[w][v]=1
|      g.nE=g.nE+1
|   end if
```

for undirected

Edge removal

Similarly opposite.

```
removeEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g without (v,w)
|
|   if g.edges[v][w] ≠ 0 then  // (v,w) in graph
|      g.edges[v][w]=0        // set to false
|      g.edges[w][v]=0
|      g.nE=g.nE-1
|   end if
```

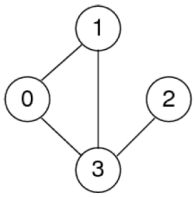
Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nV-1 do
|   |   for all j=i+1 to g.nV-1 do
|   |      if g.edges[i][j] ≠ 0 then
|   |         print i"—"j
|   |      end if
|   |   end for
|   end for
```

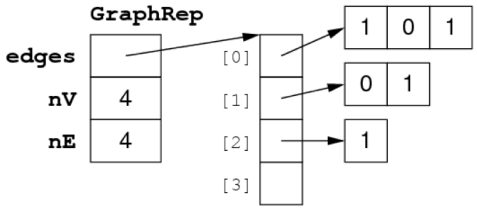
Only check top diagonal segment, since symmetric.

Optimise

A storage optimisation: store only top-right part of matrix.



Undirected graph



Only store

still $O(V^2)$ but half

New storage cost: *V-1* int ptrs + *V(V+1)/2* ints (but still $O(V^2)$)

Requires us to always use edges *(v,w)* such that *v < w*.

3. Adjacency List Representation

Cost →

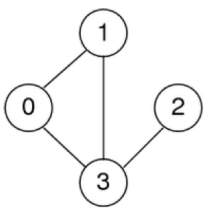
Storage cost: *O(V+E)*

Cost of operations:

- initialisation: *O(V)* (initialise *V* lists)
- insert edge: *O(E)* (need to check if vertex in list)
- delete edge: *O(E)* (need to find vertex in list)

Could sort vertex lists, but no benefit (although no extra cost)

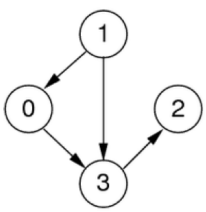
For each vertex, store linked list of adjacent vertices:



Undirected graph

A[0] = <1, 3>
A[1] = <0, 3>
A[2] = <3>
A[3] = <0, 1, 2>

Link list is storing vertices that is connecting to.



Directed graph

A[0] = <3>
A[1] = <0, 3>
A[2] = < >
A[3] = <2>

Advantages

- relatively easy to implement in languages like C
- can represent graphs and digraphs
- memory efficient if *E:V* relatively small

Disadvantages:

- one graph has many possible representations
  (unless lists are ordered by same criterion e.g. ascending)

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=newList()   // empty list
|  end for
|  return g
```

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g containing (v,w)
|
|  if not ListMember(g.edges[v],w) then
|     // (v,w) not in graph
|     ListInsert(g.edges[v],w)
|     ListInsert(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

Edge removal:

```
removeEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g without (v,w)
|
|  if ListMember(g.edges[v],w) then
|     // (v,w) in graph
|     ListDelete(g.edges[v],w)
|     ListDelete(g.edges[w],v)
|     g.nE=g.nE-1
|  end if
```

Print a list of edges

```
showEdges(g):
|  Input graph g
|
|  for all i=0 to g.nV-1 do
|  |  for all v in g.edges[i] do
|  |     if i < v then
|  |        print i"—"v
|  |     end if
|  |  end for
|  end for
```

Summary of operations above:

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
initialise	1	V^2	V
insert edge	E	1	E
remove edge	E	1	E

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	1
isPath(x,y)?	$E \cdot \log V$	V^2	$V+E$
copy graph	E	V^2	$V+E$
destroy graph	1	V	$V+E$

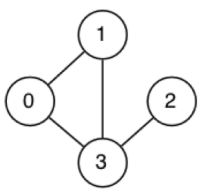
Graph ADT

Graph ADT interface **Graph.h**

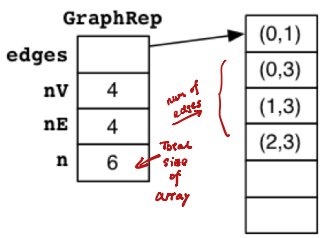
```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);  // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex);
      // is there an edge between two vertices?
void  freeGraph(Graph);
```

1. Array of Edges.

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..nV-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```



Undirected graph



Implementation of graph initialisation (array-of-edges)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));
   assert(g->edges != NULL);
   return g;
}
```

How much is enough? ... No more than *V(V-1)/2* ... Much less in practice (sparse graph)

↳ if vertices are 5
5(4)/2 = 10 edges.

Some useful utility functions:

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
}

// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

// check if an edge is valid in a graph
bool validE(Graph g, Edge e) {
   return (g != NULL && validV(e.v) && validV(e.w));
}
```

Implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n && isValidE(g,e));
   int i = 0;  // can't define in for (...)
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE)  // edge e not found
      g->edges[g->nE++] = e;
}
```

Implementation of edge removal (array-of-edges)

```
void removeEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)  // edge e found
      g->edges[i] = g->edges[--g->nE];
}
```

decrement

Implementation of edge check (array-of-edges)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));
   Edge e;
   e.v = x; e.w = y;
   for (int i = 0; i < g->nE; i++) {
      if (eq(e,g->edges[i]))  // edge found
         return true;
   }
   return false;  // edge not found
}
```

Re-implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE) { // edge e not found
      if (g->n == g->nE) {  // array full; expand
         g->edges = realloc(g->edges, 2*g->n);
         assert(g->edges != NULL);
         g->n = 2*g->n;
      }
      g->edges[g->nE++] = e;
   }
}
```

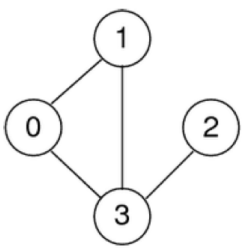
Implementation of graph removal (array-of-edges)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   free(g->edges);  // free array of edges
   free(g);         // remove Graph object
}
```

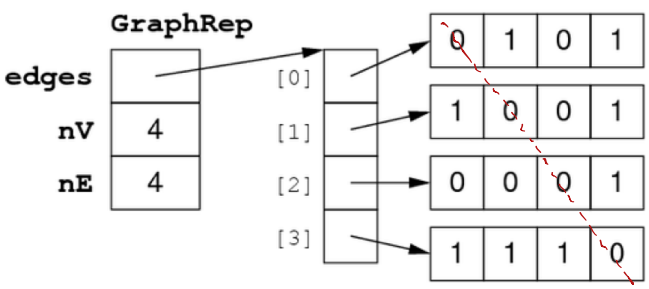
2. Adjacency Matrix (2d array)

Implementation of **`GraphRep`** (adjacency-matrix representation)

```
typedef struct GraphRep {
   int  **edges; // adjacency matrix
   int    nV;    // #vertices
   int    nE;    // #edges
} GraphRep;
```



Undirected graph



Implementation of graph initialisation (adjacency-matrix)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate array of pointers to rows
   g->edges = malloc(V * sizeof(int *));
   assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (int i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int));
      assert(g->edges[i] != NULL);
   }
   return g;
}
```

All set to be 0.

Standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size **`nelems*nbytes`**
- and sets all bytes in that block to zero

Implementation of edge insertion (adjacency-matrix)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}
```

Implementation of edge removal (adjacency-matrix)

```
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

Implementation of edge check (adjacency matrix) whether they're path to each other.

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return (g->edges[x][y] != 0);
}
```

Note: all operations, except creation, are *O(1)*

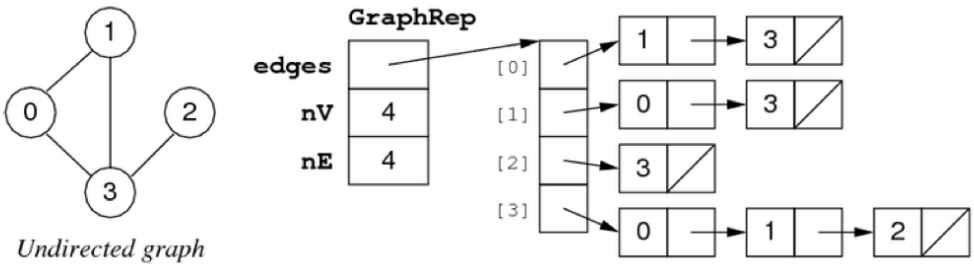
Implementation of graph removal (adjacency matrix)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      // free one row of matrix
      free(g->edges[i]);
   free(g->edges);  // free array of row pointers
   free(g);         // remove Graph object
}
```

3. Adjacency List.

Implementation of **GraphRep** (adjacency-lists representation)

```
typedef struct GraphRep {
   Node **edges;   // array of lists
   int    nV;      // #vertices
   int    nE;      // #edges
} GraphRep;
```



Assume that we have a linked list implementation

```
typedef struct Node {
   Vertex v;
   struct Node *next;
} Node;
```

with operations like **inLL**, **insertLL**, **deleteLL**, **freeLL**, e.g.

```
bool inLL(Node *L, Vertex v) {
   while (L != NULL) {
      if (L->v == v) return true;
      L = L->next;
   }
   return false;
}
```

Implementation of graph initialisation (adjacency lists)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate memory for array of lists
   g->edges = malloc(V * sizeof(Node *));
   assert(g->edges != NULL);
   for (int i = 0; i < V; i++)
      g->edges[i] = NULL;
   return g;
}
```

← Set to empty list.

Implementation of edge insertion/removal (adjacency lists)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (!inLL(g->edges[e.v], e.w)) {  // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (inLL(g->edges[e.v], e.w)) {  // edge e in graph
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

Implementation of edge check (adjacency lists)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return inLL(g->edges[x], y);
}
```

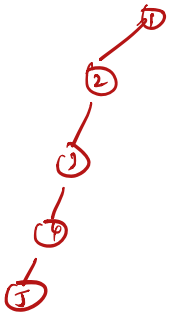
Note: all operations, except creation, are *O(E)*

Implementation of graph removal (adjacency lists)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      freeLL(g->edges[i]); // free one list
   free(g->edges);  // free array of list pointers
   free(g);         // remove Graph object
}
```

Graph Traversal

There are two strategies for graph traversal/search ...



Depth-first search (DFS)

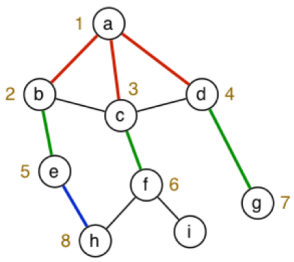
- favours following path rather than neighbours
- can be implemented recursively or iteratively (via stack)
- full traversal produces a depth-first spanning tree

Breadth-first search (BFS)

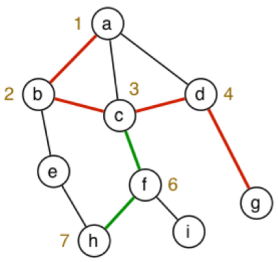
- favours neighbours rather than path following
- can be implemented iteratively (via queue)
- full traversal produces a breadth-first spanning tree

The method on the previous slide is effectively breadth-first traversal.

Comparison of BFS/DFS search for checking **`hasPath(a,h)`**



Breadth-first Search



Depth-first Search

Search will follow the size of content.

e.g.

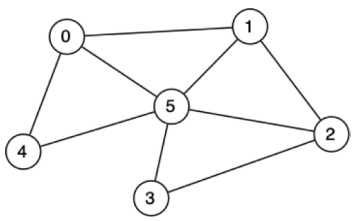
it will go c first, as b → c → e.

Both approaches ignore some edges by remembering previously visited vertices.

A spanning tree of a graph

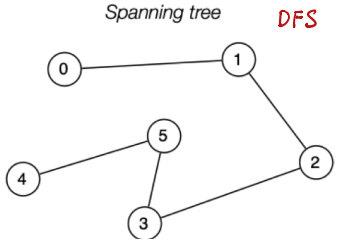
- includes all vertices, using a subset of edges, without cycles

Consider the following graph:

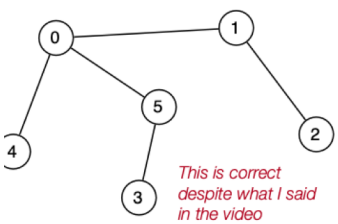


Consider how DFS and BFS could produce its spanning tree

Spanning tree DFS



Spanning tree BFS.



This is correct despite what I said in the video

Depth first search

Recursive.

Cost.

Cost analysis:

- each vertex visited at most once ⇒ cost = *O(V)*
- visit all edges incident on visited vertices ⇒ cost = *O(E)*
  - assuming an adjacency list representation

Time complexity of DFS: *O(V+E)* (adjacency list representation)

Depth-first search can be described recursively as

```
visited = {}

depthFirst(G,v):
|   visited = visited U {v}
|   for all (v,w) ∈ edges(G) do
|   |   if w ∉ visited then
|   |   |   depthFirst(G,w)
|   |   end if
|   end for
```

(graph G, node v)

Final Element.

Add node v into visited set.

starting node, ending node

Looking for neighbours

If not visited,

Do further research with w, which is ending node in previous case

The recursion induces backtracking

OR Final Path to Element.

Recursive DFS path checking

```
visited = {}

hasPath(G,src,dest):
|   Input  graph G, vertices src,dest
|   Output true if there is a path from src to dest,
|          false otherwise
|
|   return dfsPathCheck(G,src,dest)
```

Requires wrapper around recursive function **`dfsPathCheck()`**

Recursive function for path checking

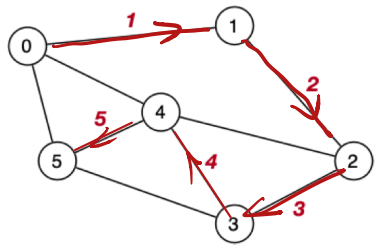
source

```
dfsPathCheck(G,v,dest):
|   visited = visited U {v}
|   for all (v,w) ∈ edges(G) do
|   |   if w=dest then    // found edge to dest
|   |      return true
|   |   else if w ∉ visited then
|   |      if dfsPathCheck(G,w,dest) then
|   |         return true  // found path via w to dest
|   |      end if
|   |   end if
|   end for
|   return false  // no path from v to dest
```

Add current node into visited.

search neighbours.

Tracing the execution of **`dfsPathCheck(G,0,5)`** on:



Reminder: we consider neighbours in ascending order

Clearly does not find the shortest path ⇒ BFS faster in this case

Recording the path for DFS by global array.

Knowing whether a path exists can be useful

Knowing what the path is, is even more useful

Strategy:

- record the previously visited node as we search
- so that we can then trace path (backwards) through graph

Requires a global array (not a set):

- **`visited[v]`** contains vertex **`w`** from which we reached **`v`**

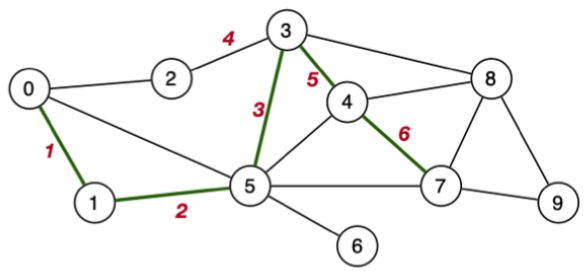
Function to find path src→dest and print it

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPath(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  visited[src]=src  // starting node of the path
|  if dfsPathCheck(G,src,dest) then
|  |  // show path in dest..src order
|  |  v=dest
|  |  while v≠src do
|  |     print v"-"
|  |     v=visited[v]
|  |  end while
|  |  print src
|  end if
```

Recursive function to build path in **`visited[]`**

```
dfsPathCheck(G,v,dest):
|  for all (v,w) ∈ edges(G) do
|  |  if visited[w] = -1 then
|  |     visited[w] = v
|  |     if w = dest then  // found edge from v to dest
|  |        return true
|  |     else if dfsPathCheck(G,w,dest) then
|  |        return true  // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false  // no path from v to dest
```

The **`visited[]`** array after **`dfsPathCheck(G,0,7)`** succeeds



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
visited	0	0	3	5	3	1	5	4	-1	-1

Actual nodes →

e.g. To get to [3] from 5.

We record where we came from once we reach a node.

Originally all "-1"

Depth first search via stack

DFS can also be described non-recursively (via a stack):

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPathDFS(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  push src onto new stack S
|  while not found ∧ S is not empty do
|  |  pop v from S
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) with visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     push w onto S
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard stack operations ... Time complexity is still *O(V+E)*

Breadth first search. nice queue

Time complexity of BFS: *O(V+E)* (same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between *src* and *dest*.
- stops with first-found path, if there are multiple one

In many applications, edges are weighted and we want path

- based on minimum sum-of-weights along path *src*.. *dest*

We discuss weighted/directed graphs later.

col

row

v

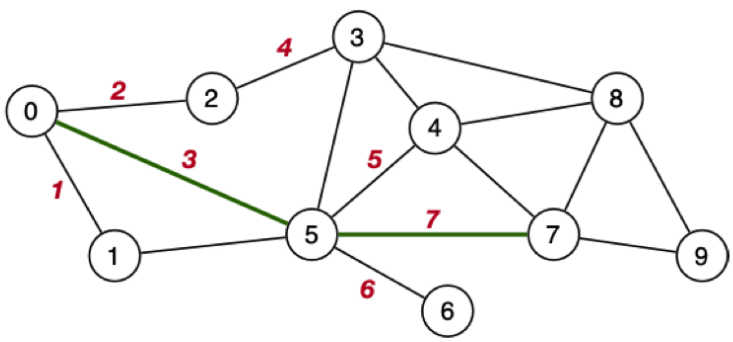
Right : col + 1
left : col - 1
up : row - 1
down : row + 1

BFS path finding algorithm:

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPathBFS(G,src,dest):
|  Input  graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  enqueue src into queue Q
|  while not found ∧ Q is not empty do
|  |  dequeue v from Q
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w) ∈ edges(G) with visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     enqueue w into Q
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

The **`visited[]`** array after **`findPathBFS(G,0,7)`** succeeds



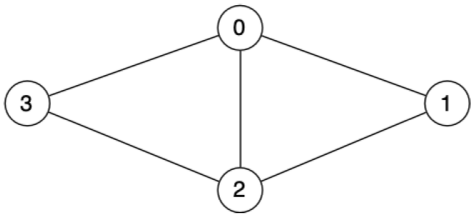
visited	0	0	0	2	5	0	5	5	-1	-1
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Graph Algorithms Intro.

A graph has a cycle if

- it has a path of length > 2
- with start vertex *src* = end vertex *dest* 首尾相连.
- and without using any edge more than once 不重复路线.

This graph has 3 distinct cycles: 0-1-2-0, 2-3-0-2, 0-1-2-3-0



("distinct" means the *set* of vertices on the path, not the order)

Cycle checking.

First attempt at checking for a cycle

```
hasCycle(G):
|   Input  graph G
|   Output true if G has a cycle, false otherwise
|
|   choose any vertex v ∈ G
|   return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
|   mark v as visited
|   for each (v,w) ∈ edges(G) do
|   |   if w has been visited then   // found cycle
|   |       return true
|   |   else if dfsCycleCheck(G,w) then
|   |       return true
|   end for
|   return false   // no cycle at v
```

← Back to beginning.

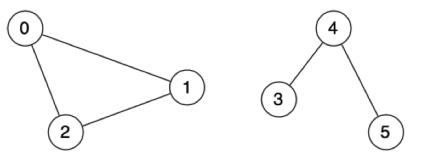
The above algorithm has two bugs ...

- only one connected component is checked
- the loop `for each (v,w) ∈ edges(G) do` should exclude the neighbour of *v* from which you just came, so as to prevent a single edge *w*-*v* being classified as a cycle.

Version of cycle checking (in C) for one connected component:

```
bool dfsCycleCheck(Graph g, Vertex v, Vertex u) {
   visited[v] = true;
   for (Vertex w = 0; w < g->nV; w++) {
      if (adjacent(g, v, w)) {
         if (!visited[w]) {
            if (dfsCycleCheck(g, w, v))
               return true;
         }
         else if (w != u)
            return true;
      }
   }
   return false;
}
```

If we start from vertex 5 in the following graph, we don't find the cycle:



Connected Component #1 *Connected Component #2*

Wrapper to ensure that all connected components are checked:

```
Vertex *visited;

bool hasCycle(Graph g, Vertex s) {
   bool result = false;
   visited = calloc(g->nV,sizeof(int));
   for (int v = 0; v < g->nV; v++) {
      for (int i = 0; i < g->nV; i++)
         visited[i] = -1;
      if dfsCycleCheck(g, v, v)) {
         result = true;
         break;
      }
   }
   free(visited);
   return result;
}
```

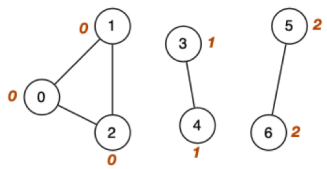
Connected component.

Consider these problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build **`componentOf[]`** array, one element for each vertex *v*
- indicating which connected component *v* is in




```
nComponents(g) = 3

componentOf[1] = 0
componentOf[5] = 2

sameComponent(3,4) = true
sameComponent(3,5) = false
sameComponent(0,6) = false
```

Algorithm to assign vertices to connected components:

```
components(G):
|   Input  graph G
|   Output componentOf[] filled for all V
|
|   for all vertices v ∈ G do
|   |   componentOf[v]=-1
|   end for
|   compID=0   // component ID
|   for all vertices v ∈ G do
|   |   if componentOf[v]=-1 then
|   |      dfsComponent(G,v,compID)
|   |      compID=compID+1
|   |   end if
|   end for
```

Hamilton

And

Ruler Question

Need

to

put

in.

Directed / Weighted Graph

Discussion so far has considered graphs as

- V = set of vertices, E = set of edges

Real-world applications require more "precision"

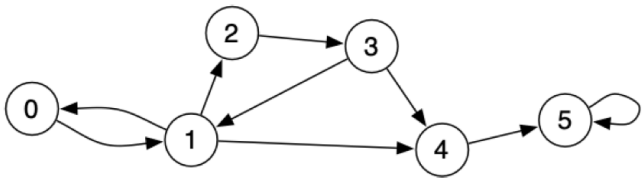
- some edges are directional (e.g. one-way streets)
- some edges have a cost (e.g. distance, traffic)

We need to consider directed graphs and weighted graphs

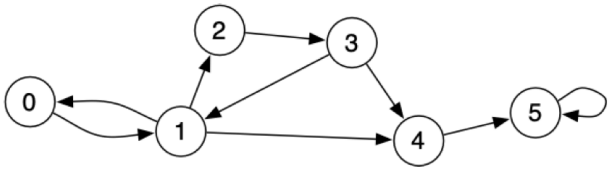
Directed graphs are ...

- graphs with V vertices, E edges (v,w)
- edge (v,w) has source v and destination w
- unlike undirected graphs, $v \rightarrow w \neq w \rightarrow v$

Example digraph:



Some properties of ...



- edges 1-2-3 form a cycle, edges 1-3-4 do *not* form a cycle
- vertex 5 has a self-referencing edge *(5,5)*
- vertices 0 and 1 reference each other, i.e. *(0,1)* and *(1,0)*
- there are no paths from 5 to any other nodes
- paths from 0→5: 0-1-2-3-4-5, 0-1-4-5, 0-1-2-3-1-4-5

Terminology for digraphs ...

Directed path: sequence of $n \geq 2$ vertices $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$

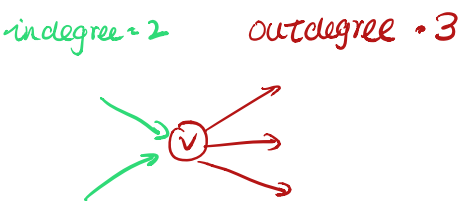
- where $(v_i,v_{i+1}) \in edges(G)$ for all v_i,v_{i+1} in sequence

If $v_1 = v_n$, we have a directed cycle

Degree of vertex: number of incident edges

- outdegree: $deg(v)$ = number of edges of the form $(v, \_)$
- indegree: $deg^{-1}(v)$ = number of edges of the form $(\_, v)$

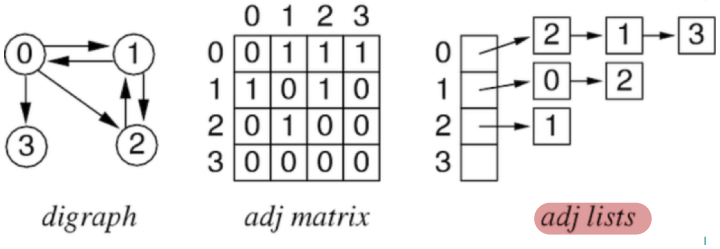
Direct cycle:



Similar set of choices as for undirectional graphs:

- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

V vertices identified by 0 .. $V-1$



Costs of representations: (where degree $deg(v)$ = #edges leaving v)

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
insert edge	E	1	1
exists edge (v,w)?	E	1	$deg(v)$
get edges leaving v	E	V	$deg(v)$

Overall, adjacency list representation is best

- real graphs tend to be sparse
  (large number of vertices, small average degree $deg(v)$)
- algorithms frequently iterate over edges from v

Weight:

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a precedence in the association (directed)

Some applications require us to consider

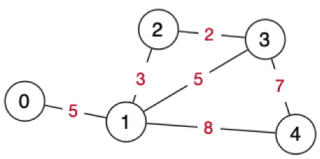
- a cost or weight of an association
- modelled by assigning values to edges (e.g. positive reals)

Weighted graphs are ...

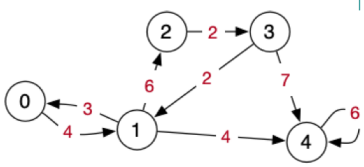
- graphs with *V* vertices, *E* edges *(s,t)*
- each edge *(s,t,w)* connects vertices *s* and *t* and has weight *w*

Weights can be used in both directed and undirected graphs.

Example weighted graphs:



Weighted Graph



Directed Weighted Graph

Weights lead to minimisation-type questions, e.g.

e.g. Telstra trying to lay cables to houses (vertices)?

1. Cheapest way to connect all vertices?

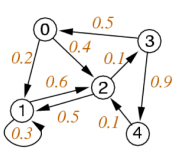
- a.k.a. minimum spanning tree problem
- assumes: edges are weighted and undirected

2. Cheapest way to get from *A* to *B*?

Google Map.

- a.k.a shortest path problem
- assumes: edge weights positive, directed or undirected

Adjacency matrix representation with weights:



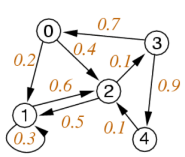
Weighted Digraph

	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

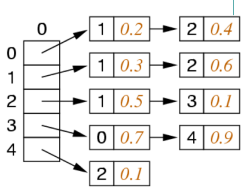
Adjacency Matrix

Note: need distinguished value to indicate "no edge".

Adjacency lists representation with weights:



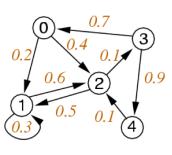
Weighted Digraph



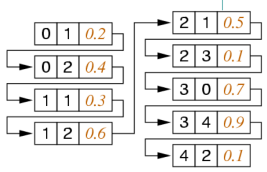
Adjacency Lists

Note: if undirected, each edge appears twice with same weight

Edge array / edge list representation with weights:



Weighted Digraph



Edge List

Note: not very efficient for use in processing algorithms, but does give a possible representation for min spanning trees or shortest paths

Implement:

Changes to preious grpah data structures to include weights:

WGraph.h

```
// edges are pairs of vertices (end-points) plus weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

Note: here, we assume all weights are positive, but not required

WGraph.c (assuming adjacency matrix representation)

```
typedef struct GraphRep {
   int **edges;  // adjacency matrix storing weights
                 // 0 if nodes not adjacent
   int   nV;     // #vertices
   int   nE;     // #edges
} GraphRep;

bool adjacent(Graph g, Vertex v, Vertex w) {
   assert(valid graph, valid vertices)
   return (g->edges[v][w] != 0);
}
```

More **WGraph.c**

```
void insertEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not already in graph
   if (g->edges[e.v][e.w] == 0) g->nE++;
   // may change weight of existing edge
   g->edges[e.v][e.w] = e.weight;
   g->edges[e.w][e.v] = e.weight;
}

void removeEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not in graph
   if (g->edges[e.v][e.w] == 0) return;
   g->edges[e.v][e.w] = 0;
   g->edges[e.w][e.v] = 0;
   g->nE--;
}
```

Directed Graph

Problems to solve on digraphs:

- is there a directed path from *s* to *t*? (transitive closure)
- what is the shortest path from *s* to *t*? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
- how to build a web crawler? (graph traversal)

Problem: computing reachability (**`reachable(G,s,t)`**)

Given a digraph *G* it is potentially useful to know

- is vertex *t* reachable from vertex *s*?

Example applications:

- can I complete a schedule from the current state?
- is a malloc'd object being referenced by any pointer?

One possibility to implement a reachability check:

- use **`hasPath(G,s,t)`** (itself implemented by DFS or BFS algorithm)
- feasible only if *reachable(G,s,t)* is an infrequent operation

What about applications that frequently check reachability?

Would be very convenient/efficient to have:

```
reachable(G,s,t)  ≡  G.tc[s][t]
```

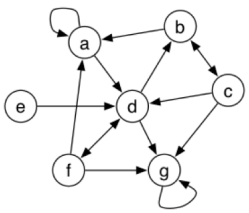
`tc[][]` is called the transitive closure matrix

- **`tc[`***s*`][`***t*`]`** is 1 if there is a path from *s* to *t*, 0 otherwise

Of course, if *V* is large, then this may not be feasible either.

The **`tc[][]`** matrix shows all directed paths in the graph

Nothing can go to e. ⇒ all 0



adjacency matrix

	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

To

From

reachability matrix

	a	b	c	d	e	f	g
a	1	1	1	1	0	1	1
b	1	1	1	1	0	1	1
c	1	1	1	1	0	1	1
d	1	1	1	1	0	1	1
e	1	1	1	1	0	1	1
f	1	1	1	1	0	1	1
g	0	0	0	0	0	0	1

g could go to nothing, except g itself

Question: how to build **`tc[][]`** from **`edges[][]`**?

How to produce reachability matrix.

Goal: produce a matrix of reachability values

Observations:

- $\forall s,t \in$ vertices(G): $(s,t) \in$ edges(G) $\Rightarrow$ *tc[s][t]* = 1
- $\forall i,s,t \in$ vertices(G): $(s,i) \in$ edges(G) $\wedge$ $(i,t) \in$ edges(G) $\Rightarrow$ *tc[s][t]* = 1

In other words

- **`tc[s][t]=1`** if there is an edge from *s* to *t* (path of length 1)
- **`tc[s][t]=1`** if there is a path from *s* to *t* of length 2 ($s \rightarrow i \rightarrow t$)

Extending the above observations gives ...

An algorithm to convert **edges** into a *tc*

```
makeTC(G):
|  tc[][] = edges[][]
|  for all i ∈ vertices(G) do
|  |  for all s ∈ vertices(G) do
|  |  |  for all t ∈ vertices(G) do
|  |  |  |  if tc[s][i]=1 ∧ tc[i][t]=1 then
|  |  |  |  |  tc[s][t]=1
|  |  |  |  end if
|  |  |  end for
|  |  end for
|  end for
```

O(n³).

s → i → t

√ s → t.

then for outside loop:

check

a → i

if

a → s ⇒ Yes

since

s → t Yes.

then

a → i Yes.

⋮

How it works ...

After copying **edges[][]**, **tc[s][t]** is 1 if $s \rightarrow t$ exists

After first iteration (**i=0**), **tc[s][t]** is 1 if

- either $s \rightarrow t$ exists or $s \rightarrow 0 \rightarrow t$ exists

vertex 0

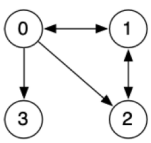
After second interation (**i=1**), **tc[s][t]** is 1 if any of

- $s \rightarrow t$ or $s \rightarrow 0 \rightarrow t$ or $s \rightarrow 1 \rightarrow t$ or $s \rightarrow 0 \rightarrow 1 \rightarrow t$ or $s \rightarrow 1 \rightarrow 0 \rightarrow t$

After the V^{th} iteration, **tc[s][t]** is 1 if

- there is a directed path in the graph from *s* to *t*

Tracing Warshall's algorithm on a simple graph:



Graph

	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	0	1	0
[2]	0	1	0	0
[3]	0	0	0	0

Initially

0 step

1 step reach

	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

After first iteration

2 steps reach.

	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

After second iteration

No change on any following iterations

Cost analysis:

- storage: additional V^2 items (but each item may be 1 bit)
- computation of transitive closure: V^3
- computation of **reachable()**: *O(1)* after generating **tc[][]**

Amortisation: need many calls to **reachable()** to justify setup cost

Alternative: use DFS in each call to **reachable()**

Cost analysis:

- storage: cost of Stack and Set during DFS calculation
- computation of **reachable()**: $O(V^2)$ (for adjacency matrix)

Same algorithms as for undirected graphs:

```
depthFirst(G,v):
|  mark v as visited
|  for each (v,w) ∈ edges(G) do
|  |  if w has not been visited then
|  |  |  depthFirst(w)
|  |  end if
|  end for

breadthFirst(G,v):
|  enqueue v
|  while queue not empty do
|  |  curr=dequeue
|  |  if curr not already visited then
|  |  |  mark curr as visited
|  |  |  enqueue each w where (curr,w) ∈ edges(G)
|  |  end if
|  end while
```

❖ Example: Web Crawling

Goal: visit every page on the web
Solution: breadth-first search with "implicit" graph

```
webCrawl(startingURL):
|  mark startingURL as alreadySeen
|  enqueue(Q,startingURL)
|  while not isEmpty(Q) do
|  |  currPage=dequeue(Q)
|  |  visit currPage
|  |  for each hyperLink on currPage do
|  |  |  if hyperLink not alreadySeen then
|  |  |     mark hyperLink as alreadySeen
|  |  |     enqueue(Q,hyperLink)
|  |  |  end if
|  |  end for
|  end while
```

`visit` scans page and collects e.g. keywords and links

❖ PageRank

Goal: determine which Web pages are "important"

Approach: ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = di-edge
- pages with many incoming hyperlinks are important
- need to computing "incoming degree" for vertices

e.g. many citation

Problem: the Web is a *very* large graph

- approx. 10^{10} pages, 10^{11} hyperlinks

Assume for the moment that we could build a graph ...

Naive PageRank algorithm:

```
PageRank(myPage):
|  rank=0
|  for each page in the Web do
|  |  if linkExists(page,myPage) then
|  |     rank=rank+1
|  |  end if
|  end for
```

Note: requires inbound link check (normally, we check outbound)

Cost:

V = # pages in Web, E = # hyperlinks in Web

Costs for computing PageRank for each representation:

Representation	linkExists(v,w)	Cost
Adjacency matrix	`edge[v][w]`	*1*
Adjacency lists	`inLL(list[v],w)`	≅ *E/V*

Not feasible ...

- adjacency matrix ... $V \cong 10^{10} \Rightarrow$ matrix has 10^{20} cells
- adjacency list ... V lists, each with ≅ 10 hyperlinks $\Rightarrow 10^{11}$ list nodes

So how to really do it?

The random web surfer strategy.

Each page typically has many outbound hyperlinks ...

- choose one at random, without a **`visited[]`** check
- follow link and repeat above process on destination page

if we visit a page very often. which means it have a lot of inbound links. then rank it.

If no visited check, need a way to (mostly) avoid loops

Important property of this strategy

- if we randomly follow links in the web ...
- ... more likely to re-discover pages with many inbound links

Random web surfer algorithm ...

```
curr=random page, prev=null
for a long time do
|   if curr not in array rank[] then
|      rank[curr]=0
|   end if
|   rank[curr]=rank[curr]+1
|   if random(0,100) < 85 then // with 85% chance ...
|      prev=curr               // ... keep crawling
|      curr=choose hyperlink from curr
|   else
|      curr=random page, not prev // avoid getting stuck
|      prev=null
|   end if
end for
```

Minimum Spanning Tree.

❖ Minimum Spanning Trees

Reminder: Spanning tree *ST* of graph *G=(V,E)*

- spanning = all vertices, tree = no cycles
- *ST* is a subgraph of *G* (*G'=(V,E')* where $E' \subseteq E$)
- *ST* is connected and acyclic

Subgraph of graph G.

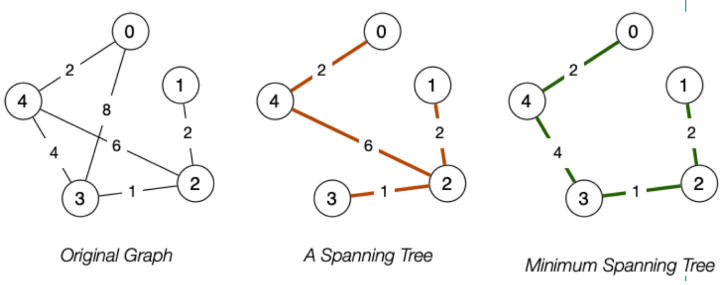
Minimum spanning tree *MST* of graph *G*

- *MST* is a spanning tree of *G*
- sum of edge weights is no larger than any other *ST*

Applications:

- Computer networks, Electrical grids, Transportation networks ...

Example:



From source to des, min path, no cycle.

How

Problem: how to (efficiently) find MST for graph *G*?

One possible strategy:

- generate all spanning trees
- calculate total weight of each
- MST = ST with lowest total weight

Note that MST may not be unique

- e.g. if all edges have same weight, then all STs are MSTs

Brute force solution (using generate-and-test strategy):

```
findMST(G):
|   Input  graph G
|   Output a minimum spanning tree of G
|
|   bestCost=∞
|   for all spanning trees t of G do
|   |   if cost(t) < bestCost then
|   |       bestTree=t
|   |       bestCost=cost(t)
|   |   end if
|   end for
|   return bestTree
```

Not useful in general because #spanning trees is potentially large

(e.g. n^{n-2} for a complete graph with n vertices)

❖ Kruskal's Algorithm

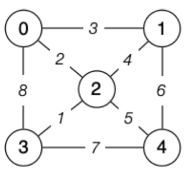
One approach to computing MST for graph *G* with *V* nodes:

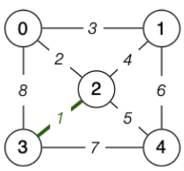
1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

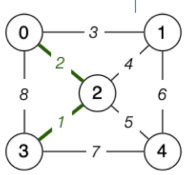
Critical operations:

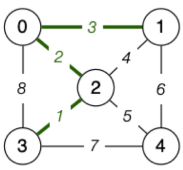
- iterating over edges in weight order
- checking for cycles in a graph

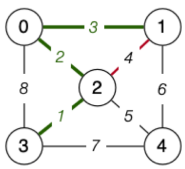
Execution trace of Kruskal's algorithm:

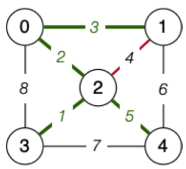












No cycle!

Now we have 4 edges 5 vertices.

Then we have a min. spanning tree.

Pseudocode:

```
KruskalMST(G):
|   Input  graph G with n nodes
|   Output a minimum spanning tree of G
|
|   MST=empty graph
|   sort edges(G) by weight
|   for each e ∈ sortedEdgeList do
|   |   MST = MST U {e}   // add edge
|   |   if MST has a cyle then
|   |      MST = MST \ {e}   // drop edge
|   |   end if
|   |   if MST has n-1 edges then
|   |      return MST
|   |   end if
|   end for
```

Simplifying assumption:

- edges in *G* are not directed (MST for digraphs is harder)

⇌

If edges are not weighted

- there is no real notion of *minimum* spanning tree

Our MST algorithms apply to

- weighted, non-directional, connected graphs

Rough time complexity analysis ...

- sorting edge list is $O(E \cdot \log E)$
- at least *V* iterations over sorted edges
- on each iteration ...
  - getting next lowest cost edge is $O(1)$
  - checking whether adding it forms a cycle: cost = $O(V^2)$

Possibilities for cycle checking:

- use DFS ... too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

❖ Prim's Algorithm

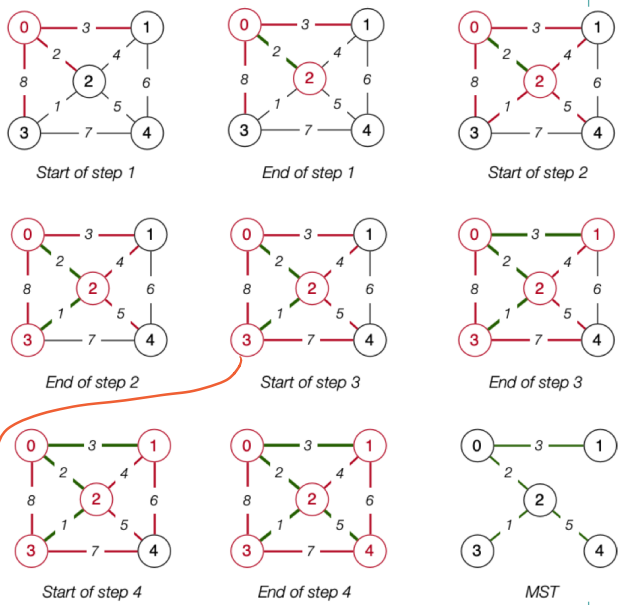
Another approach to computing MST for graph *G=(V,E)*:

1. start from any vertex *v* and empty MST
2. choose edge not already in MST to add to MST; must be:
   - incident on a vertex *s* already connected to *v* in MST
   - incident on a vertex *t* not already connected to *v* in MST
   - minimal weight of all such edges
3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

Execution trace of Prim's algorithm (starting at *s*=0):



Next opinion: 7, 8, 3, 4, 5
sort: 3, 4, 5, 7, 8.
hence, 3.

Pseudocode:

```
PrimMST(G):
|   Input  graph G with n nodes
|   Output a minimum spanning tree of G
|
|   MST=empty graph
|   usedV={0}
|   unusedE=edges(g)
|   while |usedV| < n do
|   |   find e=(s,t,w) ∈ unusedE such that {
|   |      s ∈ usedV ∧ t ∉ usedV
|   |        ∧ w is min weight of all such edges
|   |   }
|   |   MST = MST U {e}
|   |   usedV = usedV U {t}
|   |   unusedE = unusedE \ {e}
|   end while
|   return MST
```

Critical operation: finding best edge

Rough time complexity analysis ...

- *V* iterations of outer loop
- in each iteration, finding min-weighted edge ...
  - with set of edges is *O(E)* ⇒ *O(V·E)* overall
  - with priority queue is *O(log E)* ⇒ *O(V·log E)* overall

Note:

- have seen stack-based (DFS) and queue-based (BFS) traversals
- using a *priority queue* gives another non-recursive traversal

Shortest Path Algorithms

Path = sequence of edges in graph G

- $p = (v_0,v_1,weight_1), (v_1,v_2,weight_2), \ldots, (v_{m-1},v_m,weight_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p), t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Applications: navigation, routing in data networks, ...

Some variations on shortest path (SP) ...

Source-target SP problem

- shortest path from source vertex *s* to target vertex *t*

Single-source SP problem

- set of shortest paths from source vertex *s* to all other vertices

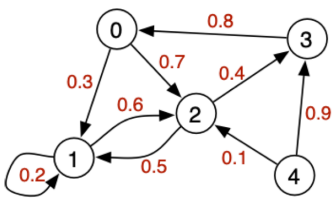
All-pairs SP problems

- set of shortest paths between all pairs of vertices *s* and *t*

Shortest paths from *s* to all other vertices

- **`dist[]`** *V*-indexed array of cost of shortest path from *s*
- **`pred[]`** *V*-indexed array of predecessor in shortest path from *s*

Example:



To:

	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

cannot reach

← Last node

Shortest paths from s=0

	0	1	2	3	4
dist	1.3	0.6	0.1	0.5	0
pred	3	2	4	2	-

Shortest paths from s=4

❖ Edge Relaxation

Assume: **`dist[]`** and **`pred[]`** as above

- but containing data for shortest paths *discovered so far*

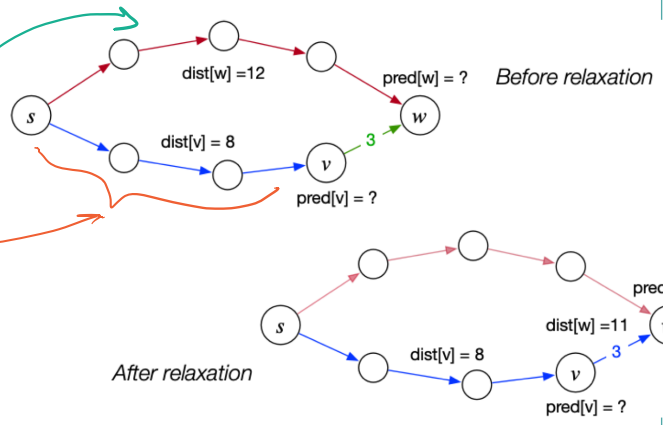
If we have ...

- **`dist[v]`** is length of shortest known path from *s* to *v*
- **`dist[w]`** is length of shortest known path from *s* to *w*
- edge *(v,w,weight)*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*:

- if **`dist[v]+weight < dist[w]`** then update **`dist[w]←dist[v]+weight`** and **`pred[w]←v`**

Relaxation along edge *e = (v,w,3)*:

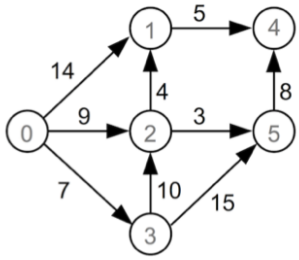


❖ Dijkstra's Algorithm

One approach to solving single-source shortest path ...

```
dist[]   // array of cost of shortest path from s
pred[]   // array of predecessor in shortest path from s
vSet     // vertices whose shortest path from s is unknown

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise all dist[] to ∞
|  dist[source]=0
|  initialise all pred[] to -1
|  vSet=all vertices of G
|  while vSet ≠ ∅ do
|  |  find v ∈ vSet with minimum dist[v]
|  |  for each (v,w,weight) ∈ edges(G) do
|  |     relax along (v,w,weight)
|  |  end for
|  |  vSet=vSet \ {v}
|  end while
```

How Dijkstra's algorithm runs when source = 0:



Initially

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	inf	inf	inf	inf	inf
pred	-	-	-	-	-	-

```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

First iteration, v=0

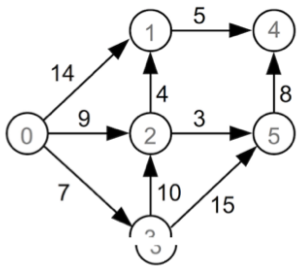
	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	int
pred	-	0	0	0	-	-

Second Iteration, v=3

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	22
pred	-	0	0	0	-	3

Third iteration, v=2

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	inf	12
pred	-	2	0	0	-	2



Fourth iteration, v=5

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	20	12
pred	-	2	0	0	5	2

```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

Fifth iteration, v=1

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Sixth iteration, v=4

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Completed, vSet is empty

Why Dijkstra's algorithm is correct ...

Hypothesis:

(a) for visited *s*, dist[*s*] is shortest distance from source

(b) for unvisited *t*, dist[*t*] is shortest distance from source *via visited nodes*

Ultimately, all nodes are visited, so ...

- ∀ *v*, dist[v] is shortest distance from source

Cost:

Time complexity analysis ...

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "**`find s ∈ vSet with minimum dist[s]`**"

1. try all **`s`**∈**`vSet`** ⇒ cost = *O(V)* ⇒ overall cost = $O(E + V^2)$ = $O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to *O(E + V·log V)*

Sorting.

Properties of sorting algorithms: stable, adaptive

Stable sort:

- let x = `a[i]`, y = `a[j]`, `key(x) == key(y)`
- "precedes" = occurs earlier in the array (smaller index)
- if x precedes y in **a**, then x precedes y in sorted **a**

Same value, whichone occur first, which will leeve in the array.

Adaptive:

This can check if the array is sorted, if sorted, which will not do Sorting algorithm.

- behaviour/performance of algorithm affected by data values
- i.e. best/average/worst case performance differs

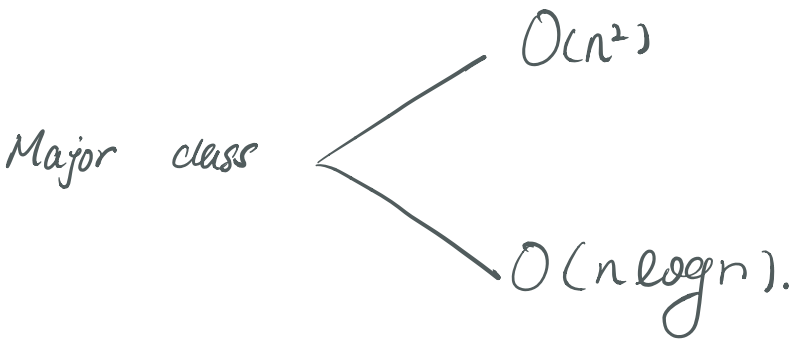
In analysing sorting algorithms:

- N = number of items = `hi-lo+1`
- C = number of comparisons between items
- S = number of times items are swapped

Aim to minimise C and S

Cases to consider for initial order of items:

- random order: `Item`s in `a[lo..hi]` have no ordering
- sorted order: `a[lo]` ≤ `a[lo+1]` ≤ ... ≤ `a[hi]`
- revserse order: `a[lo]` ≥ `a[lo+1]` ≥ ... ≥ `a[hi]`



Concrete framework:

```
// we deal with generic Items
typedef SomeType Item;

// abstractions to hide details of Items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = t;}

// Sorts a slice of an array of Items, a[lo..hi]
void sort(Item a[], int lo, int hi);

// Check for sortedness (to validate functions)
int isSorted(Item a[], int lo, int hi);
```

Implementation of the **`isSorted()`** check.

```
bool isSorted(Item a[], int lo, int hi)
{
   for (int i = lo; i < hi; i++) {
      if (!less(a[i],a[i+1])) return false;
   }
   return true;
}
```

Checks pairs (**`a[lo]`**,**`a[lo+1]`**), ... (**`a[hi-1]`**,**`a[hi]`**)

Check whole array **`Item a[N]`** via **`isSorted(a, 0, N-1)`**

The **`sort`** command

- sorts a file of text, understands fields in line
- can sort alphabetically, numerically, reverse, random

The **`qsort()`** function

- **`qsort(void *a, int n, int size, int (*cmp)())`**
- sorts any kind of array (**`n`** objects, each of **`size`** bytes)
- requires the user to supply a comparison function (e.g. **`strcmp()`**)
- sorts list of items using the order given by **`cmp()`**

Note: the comparison function is passed as a parameter; discussed elsewhere.

$O(n^2)$ Sorts:

These are adequate for small arrays.

- Selection Sort: Simple, non-adaptive.
- Bubble Sort: Simple, adaptive sort.
- Insertion Sort: Simple, adaptive sort.
- Shellsort: Improved version of insertion sort.

Selection Sort:

Simple, non-adaptive method:

- find the smallest element, put it into first array slot
- find second smallest element, put it into second array slot
- repeat until all elements are in correct position

1. Find the smallest element (swapped sth. with first element).
2. find the smallest element except for the first sorted element.

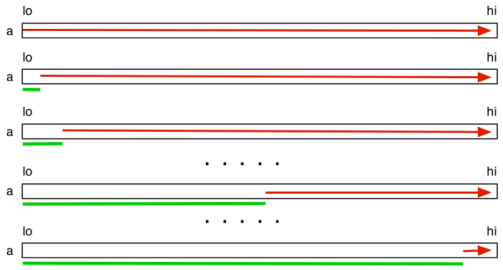
"Put in x^{th} array slot" is accomplished by:

- swapping value in x^{th} position with x^{th} smallest value

Each iteration improves "sortedness" by one element

e.g.

State of array after each iteration:



C function for Selection sort:

```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

Cost analysis (where n = **`hi-lo+1`**):

- on first pass, $n-1$ comparisons, 1 swap
- on second pass, $n-2$ comparisons, 1 swap
- ... on last pass, 1 comparison, 1 swap
- C = $(n-1)+(n-2)+...+1 = n*(n-1)/2 = (n^2-n)/2 \Rightarrow O(n^2)$

Comparison

- S = $n-1$

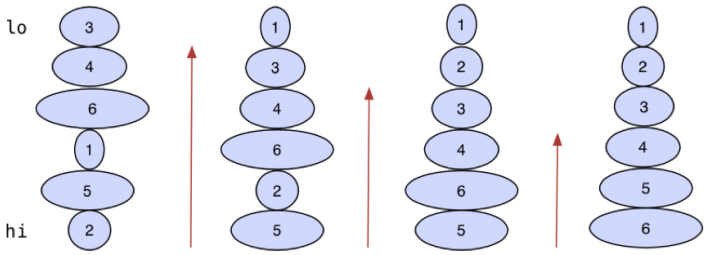
swap

Cost is same, regardless of sortedness of original array. = Non-adaptive.

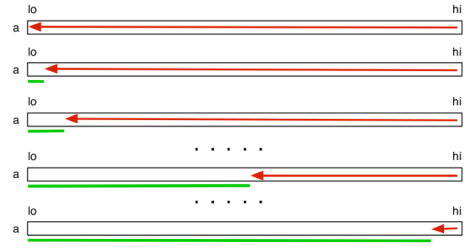
Bubble Sort.

Simple adaptive method:

- make multiple passes from *N* to *i* (*i*=0..*N*-1) from rear to front.
- on each pass, swap any out-of-order adjacent pairs
- elements move until they meet a smaller element
- eventually smallest element moves to i^{th} position
- repeat until all elements have moved to appropriate position
- stop if there are no swaps during one pass (already sorted) Adaptive Ocn's best case.



State of array after each iteration:



C function for Bubble Sort:

```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

← Adaptive.

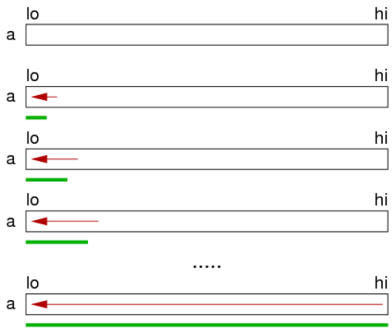
Cost analysis (where *n* = **`hi-lo+1`**):

- cost for i^{th} iteration:
  - *n-i* comparisons, ?? swaps
  - *S* depends on "sortedness", best=0, worst=*n-i*
- how many iterations? depends on data orderedness
  - best case: 1 iteration, worst case: *n-1* iterations
- $Cost_{best} = n$ (data already sorted)
- $Cost_{worst} = n-1 + ... + 1$ (reverse sorted)
- Complexity is thus $O(n^2)$

Inserting sort.

Simple adaptive method: This is not swap, is push up.

- take first element and treat as sorted array (length 1)
- take next element and insert into sorted part of array so that order is preserved
- above increases length of sorted part by one
- repeat until whole array is sorted



Insertion sort example (from Sedgewick):

e.g.

```
S O R T E X A M P L E
S O R T E X A M P L E
O S R T E X A M P L E
O R S T E X A M P L E
O R S T E X A M P L E
E O R S T X A M P L E
E O R S T X A M P L E
A E O R S T X M P L E
A E M O R S T X P L E
A E M O P R S T X L E
A E L M O P R S T X E
A E E L M O P R S T X
```

push up

tmp: E

C function for insertion sort:

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

Cost analysis (where n = **`hi-lo+1`**):

- cost for inserting element into sorted list of length i
  - C=??, depends on "sortedness", best=1, worst=i
  - S=??, don't swap, just shift, but do $C-1$ shifts
- always have n iterations
- $Cost_{best} = 1 + 1 + ... + 1$ (already sorted)
- $Cost_{worst} = 1 + 2 + ... + n = n*(n+1)/2$ (reverse sorted)
- Complexity is thus $O(n^2)$

Shellsort

Insertion sort:

- based on exchanges that only involve adjacent items
- already improved above by using moves rather than swaps
- "long distance" moves may be more efficient

shift is better than swap.
shift. → 1 assignment
swap. → 3 assignment → cost more.

Shellsort: basic idea

- array is *h*-sorted if taking every *h*'th element yields a sorted array
- an *h*-sorted array is made up of *n/h* interleaved sorted arrays
- Shellsort: *h*-sort array for progressively smaller *h*, ending with 1-sorted

3 sorted.
Array[0][3][6][9] +3 +3 +3
[1][4][7]
[2][5][8]
} Sorted.

Example *h*-sorted arrays:

	0	1	2	3	4	5	6	7	8	9
3–sorted	4	1	0	5	3	2	7	6	9	8
2–sorted	1	0	3	2	4	5	7	6	9	8
1–sorted	0	1	2	3	4	5	6	7	8	9

```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;
   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start+1; i <= hi; i++) {
         val = a[i];
         for (j = i; j >= start; j -= h) {
            if (!less(val,a[j-h])) break;
            a[j] = a[j-h];
         }
         a[j] = val;
      }
   }
}
```

Insertion sort

Effective sequences of *h* values have been determined empirically.

E.g. $h_{i+i} = 3h_i+1$ ... 1093, 364, 121, 40, 13, 4, 1

Efficiency of Shellsort:

- depends on the sequence of *h* values
- suprisingly, Shellsort has not yet been fully analysed
- above sequence has been shown to be $O(n^{3/2})$
- others have found sequences which are $O(n^{4/3})$

With slightly advantage than $O(n^2)$

Small Summary.

Comparison of sorting algorithms (animated comparison)

	#compares			#swaps			#moves		
	min	avg	max	min	avg	max	min	avg	max
Selection sort	n^2	n^2	n^2	n	n	n	.	.	.
Bubble sort	n	n^2	n^2	0	n^2	n^2	.	.	.
Insertion sort	n	n^2	n^2	.	.	.	n	n^2	n^2
Shell sort	n	$n^{4/3}$	$n^{4/3}$	.	.	.	1	$n^{4/3}$	$n^{4/3}$

more expensive →

Which is best?

- depends on cost of compare vs swap vs move for **`Item`**s
- depends on likelihood of average vs worst case

Similar in link list.

Selection sort on linked lists

- L = original list, S = sorted list (initially empty)
- find largest value V in L; unlink it
- link V node at front of S

Bubble sort on linked lists

- traverse list: if current > next, swap node values
- repeat until no swaps required in one traversal

Selection sort on linked lists

- L = original list, S = sorted list (initially empty)
- scan list L from start to finish
- insert each item into S in order

Quick sort. O(nlogn).

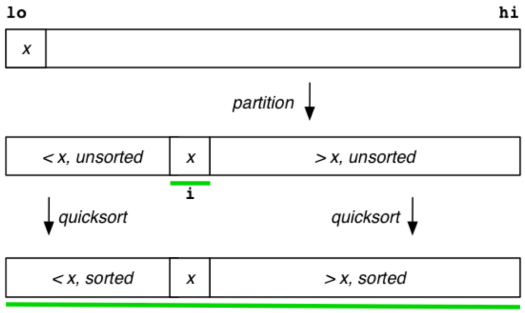
Previous sorts were all $O(n^k)$ (where $k>1$).

We can do better ...

Quicksort: basic idea

- choose an item to be a "pivot"
- re-arrange (partition) the array so that
  - all elements to left of pivot are smaller than pivot
  - all elements to right of pivot are greater than pivot
- (recursively) sort each of the partitions

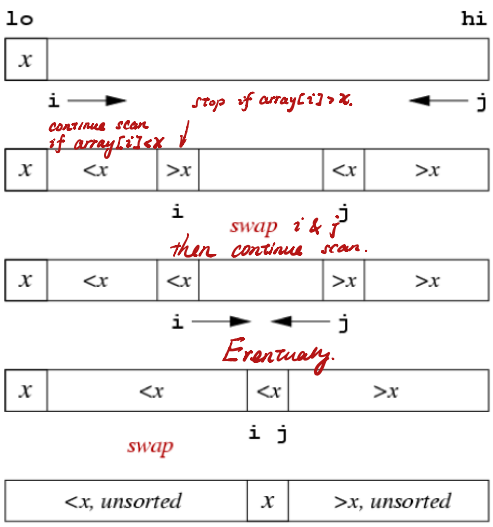
Phases of quicksort:



Elegant recursive solution ...

```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Partitioning phase:



Partition implementation:

```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];  // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

But it is not efficient.

Best case: $O(n\log n)$ comparisons

We want this! So!

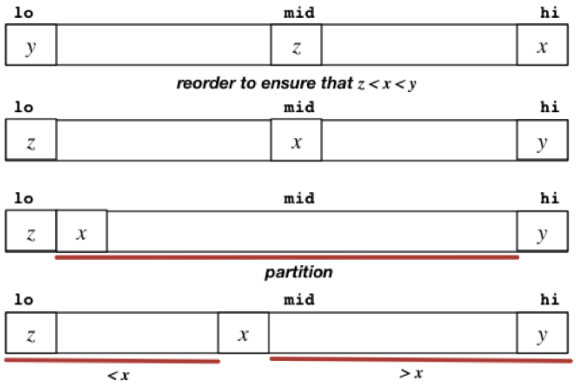
- choice of pivot gives two equal-sized partition
- same happens at every recursive level
- each "level" requires approx n comparisons
- halving at each level $\Rightarrow \log_2 n$ levels

Worst case: $O(n^2)$ comparisons

- always choose lowest/highest value for pivot
- partitions are size 1 and $n-1$
- each "level" requires approx n comparisons
- partitioning to 1 and $n-1$ $\Rightarrow n$ levels

Choice of pivot can have significant effect:

- always choosing largest/smallest ⇒ worst case
- try to find "intermediate" value by median-of-three



Median-of-three partitioning:

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   // now, we have a[lo] < a[mid] < a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, mid, lo+1);
}
void quicksort(Item a[], int lo, int hi)
{
   if (hi <= lo) return;
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Partition implementation:

```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];  // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

then v is median value ✓

Another improvement:

Another source of inefficiency:

- pushing recursion down to very small partitions
- overhead in recursive function calls
- little benefit from partitioning when size < 5

→ Do simple sorting algo instead.

Solution: handle small partitions differently

- switch to insertion sort on small partitions, or
- don't sort yet; use post-quicksort insertion sort

Then eventually:

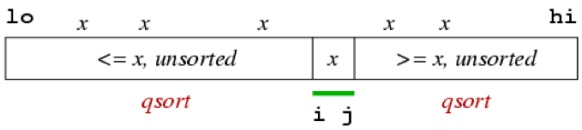
Quicksort with thresholding ...

```
void quicksort(Item a[], int lo, int hi)
{
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
      return;
   }
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

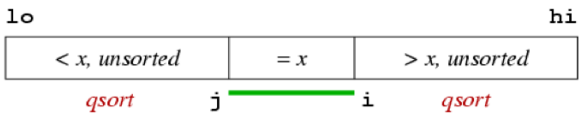
If there is multiple x in the array.

If the array contains many duplicate keys

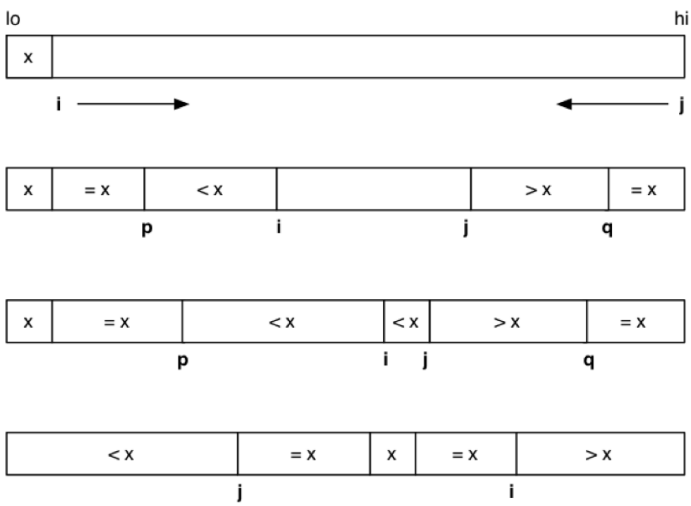
- standard partitioning does not exploit this



- can improve performance via three-way partitioning



Bentley/McIlroy approach to three-way partition:



Quicksort can be implemented using an explicit stack:

```
void quicksortStack (Item a[], int lo, int hi)
{
   Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s);
      hi = StackPop(s);
      if (hi > lo) {
         int i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
         StackPush(s,i-1); StackPush(s,lo);
      }
   }
}
```

Mergesort (Didn't understand).

Mergesort: basic idea

- split the array into two equal-sized paritions
- (recursively) sort each of the partitions
- merge the two partitions into a new sorted array
- copy back to original array

Merging: basic idea

- copy elements from the inputs one at a time
- give preference to the smaller of the two
- when one exhausted, copy the rest of the other

Best case: *O(NlogN)* comparisons

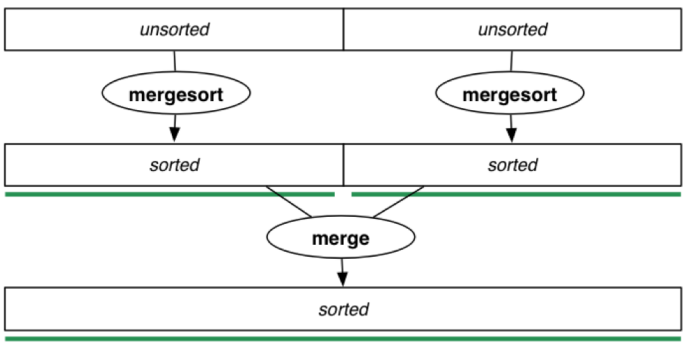
- split array into equal-sized partitions
- same happens at every recursive level
- each "level" requires $\leq N$ comparisons
- halving at each level $\Rightarrow log_2 N$ levels

Worst case: *O(NlogN)* comparisons (Better than O(n) Compare to quicksort.)

- partitions are exactly interleaved
- need to compare all the way to end of partitions

Disadvantage over quicksort: need extra storage *O(N)*

Phases of mergesort



Mergesort function:

```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

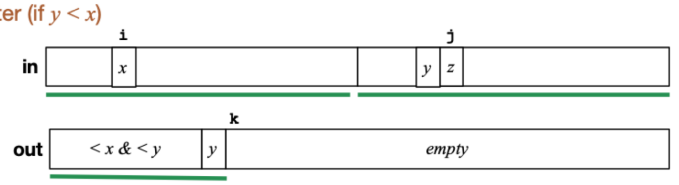
Example of use (**`typedef int Item`**):

```
int nums[10] = {32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);
```

One step in the merging process:

Before

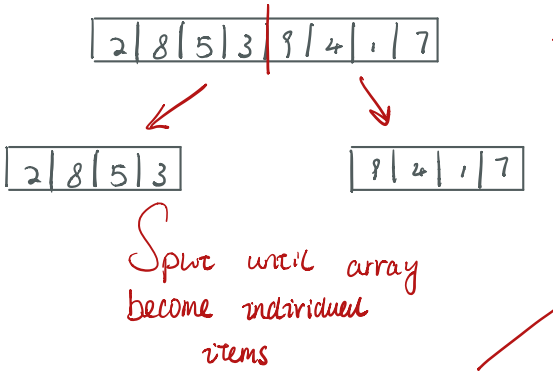
After (if $y < x$)



Implementation of merge:

```
void merge(Item a[], int lo, int mid, int hi)
{
   int  i, j, k, nitems = hi-lo+1;
   Item *tmp = malloc(nitems*sizeof(Item));

   i = lo; j = mid+1; k = 0;
   // scan both segments, copying to tmp
   while (i <= mid && j <= hi) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     else
        tmp[k++] = a[j++];
   }
   // copy items from unfinished segment
   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];

   //copy tmp back to main array
   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

Example :



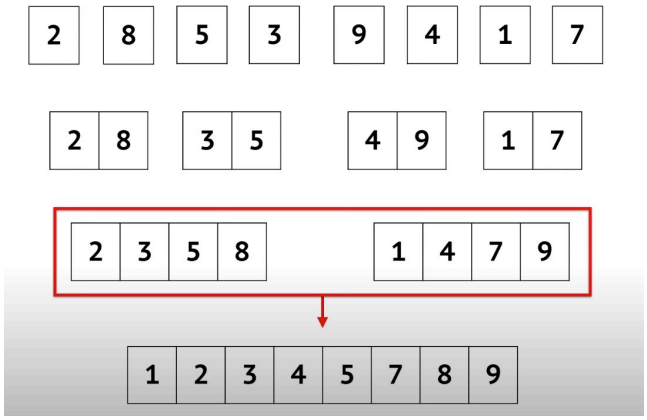
Mergesort function:

```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

Example of use (**typedef int Item**):

```
int nums[10] = {32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);
```

Then we do merge( ).



Implementation of merge:

```
void merge(Item a[], int lo, int mid, int hi)
{
   int  i, j, k, nitems = hi-lo+1;
   Item *tmp = malloc(nitems*sizeof(Item));

   i = lo; j = mid+1; k = 0;
   // scan both segments, copying to tmp
   while (i <= mid && j <= hi) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     else
        tmp[k++] = a[j++];
   }
   // copy items from unfinished segment
   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];

   //copy tmp back to main array
   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

❖ Bottom-up Mergesort

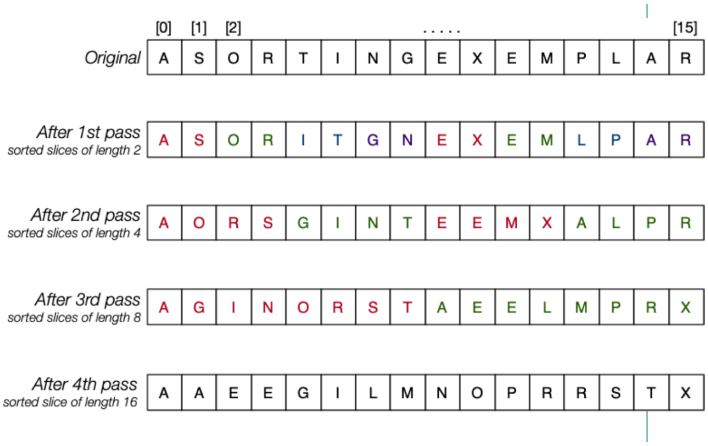
Non-recursive mergesort does not require a stack

- partition boundaries can be computed iteratively

Bottom-up mergesort:

- on each pass, array contains sorted runs of length *m*
- at start, treat as *N* sorted runs of length 1
- 1st pass merges adjacent elements into runs of length 2
- 2nd pass merges adjacent 2-runs into runs of length 4
- continue until a single sorted run of length *N*

This approach can be used for "in-place" mergesort.



Normal Mergesort:
- Break whole into part.
- Form part into sorted whole.

Bottom-up Mergesort:
- Sort whole into sorted whole.

Bottom-up mergesort for arrays:

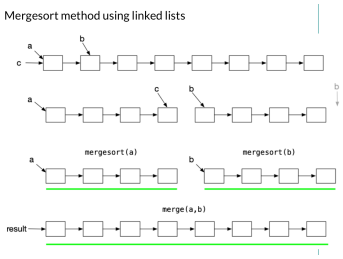
```
#define min(A,B) ((A < B) ? A : B)

void mergesort(Item a[], int lo, int hi)
{
   int m;     // m = length of runs
   int len;   // end of 2nd run
   Item c[]; // array to merge into
   for (m = 1; m <= lo-hi; m = 2*m) {
      for (int i = lo; i <= hi-m; i += 2*m) {
         len = min(m, hi-(i+m)+1);
         merge(&a[i], m, &a[i+m], len, c);
         copy(&a[i], c, m+len);
      }
   }
}
```

```
// merge arrays a[] and b[] into c[]
// aN = size of a[], bN = size of b[]
void merge(Item a[], int aN, Item b[], int bN, Item c[])
{
   int i; // index into a[]
   int j; // index into b[]
   int k; // index into c[]
   for (i = j = k = 0; k < aN+bN; k++) {
      if (i == aN)
         c[k] = b[j++];
      else if (j == bN)
         c[k] = a[i++];
      else if (less(a[i],b[j]))
         c[k] = a[i++];
      else
         c[k] = b[j++];
   }
}
```

Similar in Link List:

Merging linked lists is relatively straightforward:

```
List merge(List a, List b)
{
   List new = newList();
   while (a != NULL && b != NULL) {
      if (less(a->item, b->item))
         { new = ListAppend(new, a->item); a = a->next; }
      else
         { new = ListAppend(new, b->item); b = b->next; }
   }
   while (a != NULL)
      { new = ListAppend(new, a->item); a = a->next; }
   while (b != NULL)
      { new = ListAppend(new, b->item); b = b->next; }
   return new;
}
```



Recursive linked list mergesort, built with list merge:

```
List mergesort(List c)
{
   List a, b;
   if (c == NULL || c->next == NULL) return c;
   a = c;  b = c->next;
   while (b != NULL && b->next != NULL)
      { c = c->next; b = b->next->next; }
   b = c->next; c->next = NULL;  // split list
   return merge(mergesort(a), mergesort(b));
}
```

Heapsort.

Heapsort uses a priority queue (PQ) implemented as a heap.

Reminder: heap is a top-to-bottom ordered tree

- that has a simple implementation as an array of **`Item`**s

Reminder: priority queues ...

- implement a key-ordered queue structure
- items added to queue in arrival order
- items removed from queue in max-first order

Heapsort (really PQ-sort) approach:

- insert all array items into priority queue
- one-by-one, remove all items from priority queue
- inserting each into successive array element

Priority queue operations ...

```
PQueue newPQueue();
void PQJoin(PQueue q, Item it);
Item PQLeave(PQueue q); // remove max Item
int  PQIsEmpty(PQueue q);
```

Implementation of HeapSort:

```
void HeapSort(Item a[], int lo, int hi)
{
   PQueue pq = newPQueue();
   int i;
   for (i = lo; i <= hi; i++) {
      PQJoin(pq, a[i]);
   }
   for (i = hi; i >= lo; i--) {
      Item it = PQLeave(pq);
      a[i] = it;
   }
}
```

Problem: requires an additional data structure (*O(N)* space)

Recall that earlier we defined **`fixDown()`**

- forces value at a[k] into correct position in heap

Allowed us to work with arrays as heap structures, hence as PQs.

Can we use these ideas to build an in-array PQ-sort?

Reminder: **`fixDown()`** function

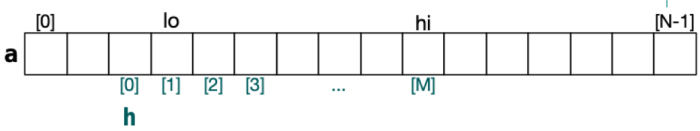
```
// force value at a[i] into correct position in a[1..N]
// note that N gives max index *and* number of items
void fixDown(Item a[], int i, int N)
{
   while (2*i <= N) {
      // compute address of left child
      int j = 2*i;
      // choose larger of two children
      if (j < N && less(a[j], a[j+1])) j++;
      if (!less(a[i], a[j])) break;
      swap(a, i, j);
      // move one level down the heap
      i = j;
   }
}
```

Heapsort: multiple iterations over a shrinking heap

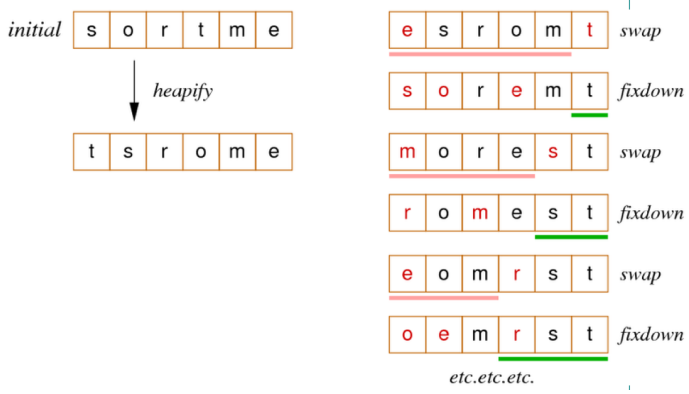
- initially use whole array as a heap
- uses **`fixDown`** to set max value at end
- reduce size of heap, and repeat

One minor complication: **`a[lo..hi]`** vs **`h[1..M]`** (where **`M=hi-lo+1`**)

To solve: pretend that heap starts one location earlier.



Trace of heapsort:



Heapsort algorithm:

```
void heapsort(Item a[], int lo, int hi)
{
   int i, N = hi-lo+1;
   Item *h = a+lo-1;  //start addr of heap
   // construct heap in a[0..N-1]
   for (i = N/2; i > 0; i--)
      fixDown(h, i, N);
   // use heap to build sorted array
   while (N > 1) {
      // put largest value at end of array
      swap(h, 1, N);
      // heap size reduced by one
      N--;
      // restore heap property after swap
      fixDown(h, 1, N);
   }
}
```

Cost

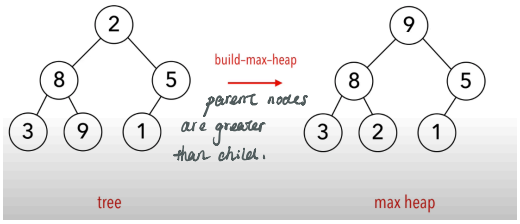
Heapsort involves two stages

- build a heap in the array
  - iterates N/2 times, each time doing **`fixDown()`**
  - each **`fixDown()`** is *O(logN)*, so overall *O(NlogN)*
  - note: can write *heapify* more efficiently than we did *O(N)*
  - note: each **`fixDown()`** involves at most $log_2(2C + S)$
- use heap to build sorted array
  - iterates N times, each time doing **`swap()`** and **`fixDown()`**
  - **`swap()`** is *O(1)*, **`fixDown()`** is *O(logN)*, so overall *O(NlogN)*

Cost of heapsort = *O(NlogN)*

sample.

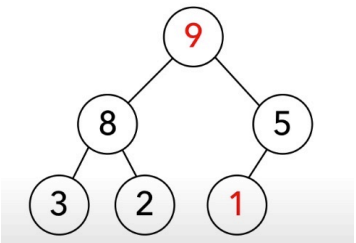
① | 2 | 8 | 5 | 3 | 9 | 1 | → | 9 | 8 | 5 | 3 | 2 | 1 |



②

| 9 | 8 | 5 | 3 | 2 | 1 |

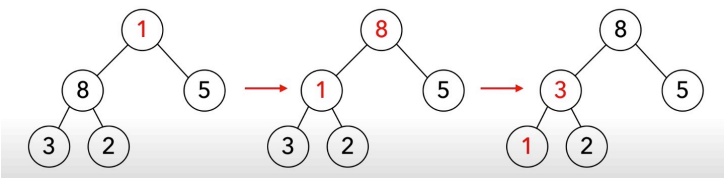
Swap 1 & 9.



③ heapify.

| 1 | 8 | 5 | 3 | 2 | 9 |

| 8 | 3 | 5 | 1 | 2 | 9 |

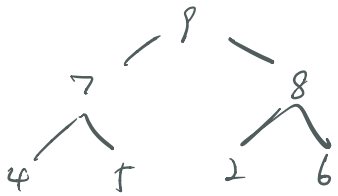


e.g.

Insert 9:

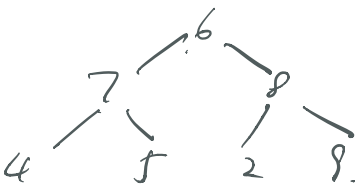
8 7 6 4 5 2 9

①.



9 7 8 4 5 2 6.

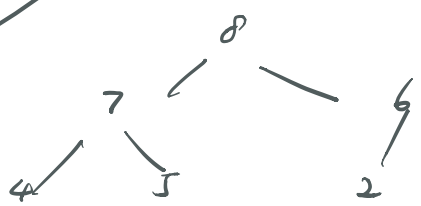
②. 6 7 8 4 5 2 9.



③

Smallest goes down
Biggest goes up.

↳ This is Max heap.
repeat step ①.



8 7 6 4 5 2 9

Summary.

Elementary sorting algorithms: $O(N^2)$ comparisons

- selection sort, insertion sort, bubble sort

Advanced sorting algorithms: $O(N \log N)$ comparisons

- quicksort, merge sort, heap sort (priority queue)

Most are intended for use in-memory (random access data structure).

Merge sort adapts well for use as disk-based sort.

Other properties of sort algorithms: stable, adaptive

Always same time cost in < best / worst.

Yes. scan → sorted → stop.

No. sort.

Selection sort:

- stability depends on implementation
- not adaptive

Bubble sort:

- is stable if items don't move past same-key items
- adaptive if it terminates when no swaps

Insertion sort:

- stability depends on implementation of insertion
- adaptive if it stops scan when position is found

Quicksort:

- easy to make stable on lists; difficult on arrays
- can be adaptive depending on implementation

Merge sort:

- is stable if merge operation is stable
- can be made adaptive (but version in slides is not)

Heap sort:

- is not stable because of top-to-bottom nature of heap ordering
- adaptive variants of heap sort exist (faster if data almost sorted)

Radix sort (Faster than O(logn) => comparative algo).

Radix sort is a non-comparative sorting algorithm.

Requires us to consider a key as a tuple $(k_1, k_2, ..., k_m)$, e.g.

- represent key 372 as (3, 7, 2)
- represent key **"sydney"** as (s, y, d, n, e, y)

Assume only small number of possible values for k_i, e.g.

- numeric: 0-9 ... alpha: a-z

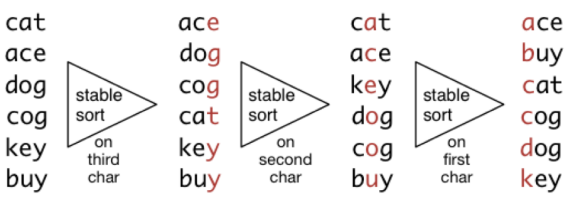
If keys have different lengths, pad with suitable character, e.g.

- numeric: 123, 002, 015 ... alpha: **"abc"**, **"zz␣"**, **"t␣␣"**

Radix sort algorithm:

- stable sort on k_m,
- then stable sort on $k_{(m-1)}$,
- continue until we reach k_1

Example:



Example:

- **m** = 3, alphabet = {'a', 'b', 'c'}, B[ ] = buckets
- A[ ] = {"abc", "cab", "baa", "a␣␣", "ca␣"}

After first pass (**i** = 3):

- B['a'] = {"baa"}, B['b'] = {"cab"}, B['c'] = {"abc"}, B['␣'] = {"a␣␣","ca␣"}
- A[ ] = {"baa", "cab", "abc", "a␣␣", "ca␣"}

After second pass (**i** = 2):

- B['a'] = {"baa","cab","ca␣"}, B['b'] = {"abc"}, B['c'] = {}, B["␣"] = {"a␣␣"}
- A[ ] = {"baa", "cab", "ca␣", "abc", "a␣␣"}

After third pass (**i** = 1):

- B['a'] = {"abc","a␣␣"}, B['b'] = {"baa"}, B['c'] = {"cab","ca␣"}, B["␣"] = {}
- A[ ] = {"abc", "a␣␣", baa", "cab", "ca␣"}

Stable sorting (bucket sort):

```
// sort array A[n] of keys
// each key is m symbols from an "alphabet"
// array of buckets, one for each symbol
for each i in m .. 1 do   right to left.
   empty all buckets
   for each key in A do
      append key to bucket[key[i]]
   end for
   clear A
   for each bucket in order do
      for each key in bucket do
         append to array
      end for
   end for
end for
```

Complexity analysis:

- array contains *n* keys, each key contains *m* symbols
- stable sort (bucket sort) runs in time *O(n)*
- radix sort uses stable sort *m* times

So, time complexity for radix sort = *O(mn)*

Radix sort performs better than comparison-based sorting algorithms

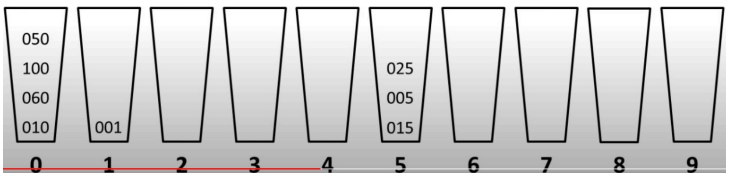
- when keys are short (small *m*) and arrays are large (large *n*)

sample

Pass 1:

010, 015, 001, 060, 005, 100, 025, 050

Pass 1 is complete.
Take the numbers out from bucket 0 to 9 from bottom



Then take everything out from left to right, bottom to top:

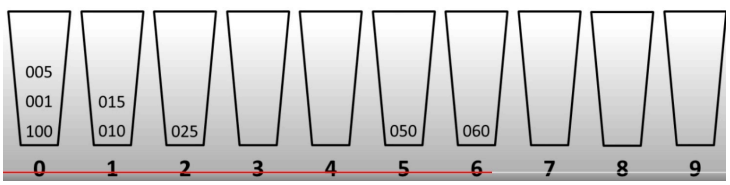
010 060 100 050 001 015 005 025

Repeat for Pass 2:

Pass 2:

010, 060, 100, 050, 001, 015, 005, 025

Pass 2 is complete.
Take the numbers out from bucket 0 to 9 from bottom



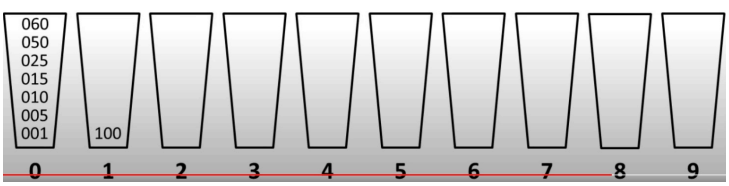
100 001 005 010 015 025 050 060

Repeat for pass 3.

Pass 3:

100, 001, 005, 010, 015, 025, 050, 060

Pass 3 is complete.
Take the numbers out from bucket 0 to 9 from bottom



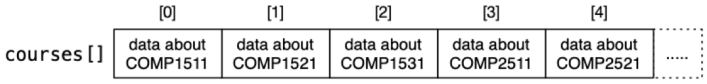
001 005 010 015 025 050 060 100. ✓ sorted.

Hashing

Associative Indexing

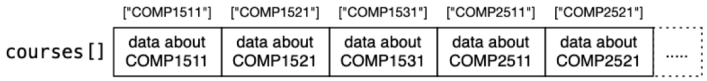
Difference between positional and associative indexing:

Positional (normal) indexing



`courses[4]` *gives access to COMP2521 data*

Associative indexing



`courses["COMP2521"]` *gives access to COMP2521 data*

How to implement?

Hashing allows us to get close to associative indexing

Ideally, we would like ...

```
courses["COMP3311"] = "Database Systems";
printf("%s\n", courses["COMP3311"]);
```

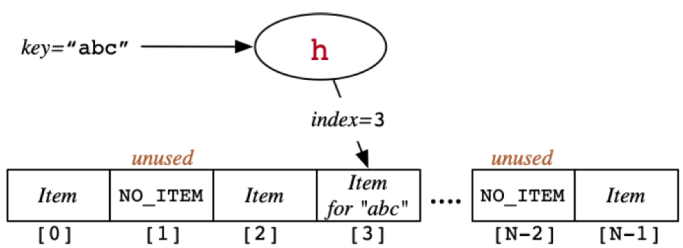
but C doesn't have non-integer index values.

Something almost as good:

```
courses[h("COMP3311")] = "Database Systems";
printf("%s\n", courses[h("COMP3311")]);
```

Hash function `h` converts key to integer and uses that as the index.

What the `h` (hash) function does



Converts a key value (of any type) into an integer index.

Sounds good ... in practice, not so simple ...

Reminder: what we'd like ...

```
courses[h("COMP3311")] = "Database Systems";
printf("%s\n", courses[h("COMP3311")]);
```

In practice, we do something like ...

```
key = "COMP3311";
item = {"COMP3311","Database Systems",...};
courses = HashInsert(courses, key, item);
printf("%s\n", HashGet(courses, "COMP3311"));
```

Advantage:

- No search needed.

To use arbitrary values as keys, we need ...

- set of **Key** values *dom(**Key**)*, each key identifies one **Item**
- an array (of size *N*) to store **Item**s
- a hash function *h()* of type *dom(**Key**)* → [0..*N*-1]
  - requirement: if *(x = y)* then *h(x) = h(y)*
  - requirement: *h(x)* always returns same value for given *x*

A problem: array is size *N*, but *dom(**Key**)* ≫ *N*

So, we also need a collision resolution method

- collision = (*x* ≠ *y* *but* *h(x) = h(y)*)

ADT:

Generalised ADT for a collection of **Item**s

Interface:

```
typedef struct CollectionRep *Collection;

Collection newCollection();     // make new empty collection
Item *search(Collection, Key); // find item with key
void insert(Collection, Item); // add item into collection
void delete(Collection, Key);  // drop item with key
```

Implementation:

```
typedef struct CollectionRep {
    ... some data structure to hold multiple Items ...
} CollectionRep;
```

For hash tables, make one change to "standard" Collection interface:

```
typedef struct HashTabRep *HashTable;
// make new empty table of size N
HashTable newHashTable(int);
// add item into collection
void HashInsert(HashTable, Item);
// find item with key
Item *HashGet(HashTable, Key);
// drop item with key
void HashDelete(HashTable, Key);
// free memory of a HashTable
void dropHashTable(HashTable);
```

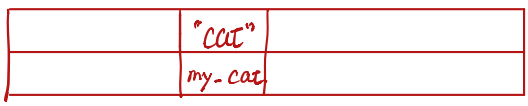
Max size.

i.e. we specify max # elements that can be stored in the collection

Example hash table implementation:

```
typedef struct HashTabRep {
   Item **items; // array of (Item *)
   int  N;       // size of array
   int  nitems;  // # Items in array
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTable new = malloc(sizeof(HashTabRep));
   new->items = malloc(N*sizeof(Item *));
   new->N = N;
   new->nitems = 0;
   for (int i = 0; i < N; i++) new->items[i] = NULL;
   return new;
}
```

① Create hash table.



Generate cat a random position.
copy the data in.

Hash table implementation (cont)

```
void HashInsert(HashTable ht, Item it) {
   int h = hash(key(it), ht->N);
   // assume table slot empty!?
   ht->items[h] = copy(it);
   ht->nitems++;
}
Item *HashGet(HashTable ht, Key k) {
   int h = hash(k, ht->N);
   Item *itp = ht->items[h];
   if (itp != NULL && equal(key(*itp),k))
      return itp;
   else
      return NULL;
}
```

enter key.
convert into index.
return pointer.

key() and **copy()** come from **Item** type; **equal()** from **Key** type

```
void HashDelete(HashTable ht, Key k) {
   int h = hash(k, ht->N);
   Item *itp = ht->items[h];
   if (itp != NULL && equal(key(*itp),k)) {
      free(itp);
      ht->items[h] = NULL;
      ht->nitems--;
   }
}
void dropHashTable(HashTable ht) {
   for (int i = 0; i < ht->N; i++) {
      if (ht->items[i] != NULL) free(ht->items[i]);
   }
   free(ht);
}
```

Characteristics of hash functions:

- converts **Key** value to index value [0..N-1]
- deterministic (key value *k* always maps to same value)
- use **mod** function to map hash value to index value
- spread key values **uniformly** over address range
  (assumes that keys themselves are uniformly distributed)
- as much as possible, $h(k) \neq h(j)$ if $j \neq k$
- cost of computing hash function must be cheap

hash function.

Basic mechanism of hash functions:

```
int hash(Key key, int N)
{
   int val = convert key to 32-bit int;
   return val % N;
}
```

If keys are **int**s, conversion is easy (identity function)

How to convert keys which are strings? (e.g. "**COMP1927**" or "**John**")

Definitely prefer that *hash*("**cat**",N) ≠ *hash*("**dog**",N)

Prefer that *hash*("**cat**",N) ≠ *hash*("**act**",N) ≠ *hash*("**tac**",N)

Universal hashing uses entire key, with position info:

```
int hash(char *key, int N)
{
   int h = 0, a = 31415, b = 21783;
   char *c;
   for (c = key; *c != '\0'; c++) {
      a = a*b % (N-1);
      h = (a * h + *c) % N;
   }
   return h;
}
```

Has some similarities with RNG. Aim: "spread" hash values over [0..N-1]

A real hash function (from PostgreSQL DBMS):

```
hash_any(unsigned char *k, register int keylen, int N)
{
   register uint32 a, b, c, len;
   // set up internal state
   len = keylen;
   a = b = 0x9e3779b9;
   c = 3923095;
   // handle most of the key, in 12-char chunks
   while (len >= 12) {
      a += (k[0] + (k[1] << 8) + (k[2] << 16) + (k[3] << 24));
      b += (k[4] + (k[5] << 8) + (k[6] << 16) + (k[7] << 24));
      c += (k[8] + (k[9] << 8) + (k[10] << 16) + (k[11] << 24));
      mix(a, b, c);
      k += 12; len -= 12;
   }
   // collect any data from remaining bytes into a,b,c
   mix(a, b, c);
   return c % N;
}
```

Where **`mix`** is defined as:

```
#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8);  \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12); \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5);  \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}
```

i.e. scrambles all of the bits from the bytes of the key value

❖ Problems with Hashing

In ideal scenarios, search cost in hash table is *O(1)*.

Rather than traversing O(n).

Problems with hashing:

- hash function relies on size of array (⇒ can't expand)
  - changing size of array effectively changes the hash function
  - if change array size, then need to re-insert all **`Item`**s
- items are stored in (effectively) random order
- if size(KeySpace) ≫ size(IndexSpace), collisions inevitable
  - collision: k ≠ j && hash(k,N) = hash(j,N)
- if **`nitems`** > **`nslots`**, collisions inevitable

Hash Collision.

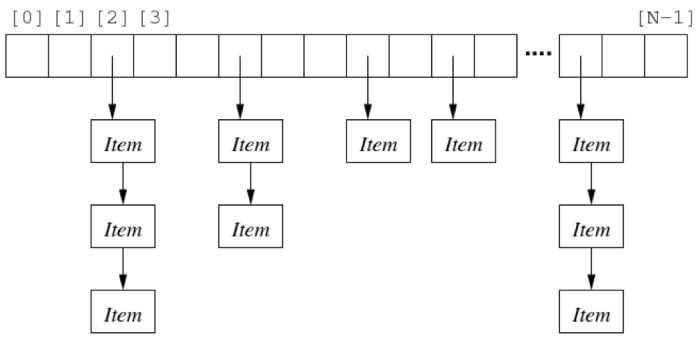
Three approaches to dealing with hash collisions:

- allow multiple **`Item`**s at a single array location
  - e.g. array of linked lists (but worst case is *O(N)*)
- systematically compute new indexes until find a free slot
  - need strategies for computing new indexes (aka probing)
- increase the size of the array
  - needs a method to "adjust" **`hash()`** (e.g. linear hashing)

①.

Solve collisions by having multiple items per array entry.

Make each element the start of linked-list of Items.



All items in a given list have the same **`hash()`** value

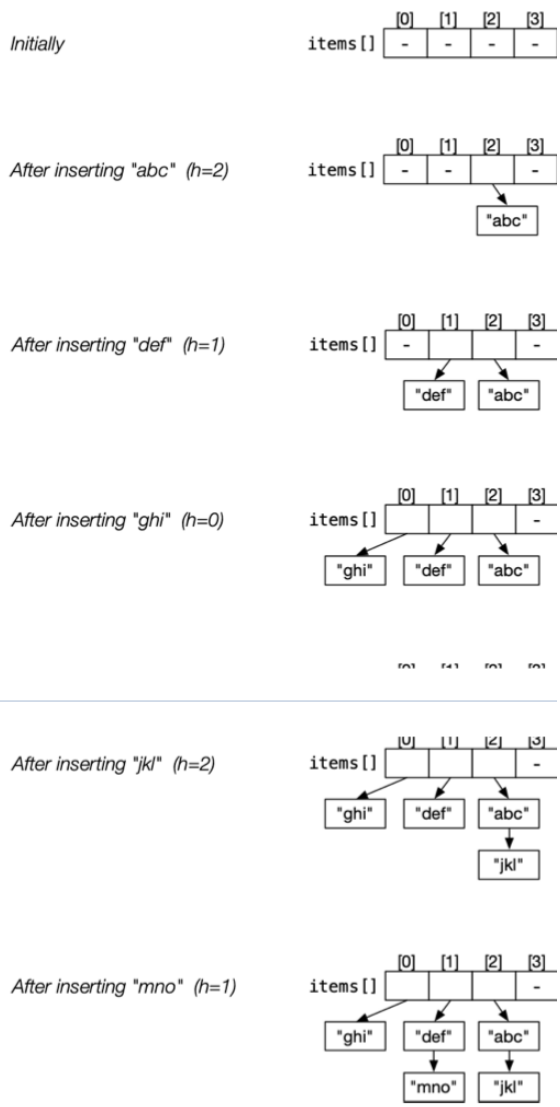
But how could we get it back?

Ans: Loop through the link list at specified position. List Search (...)

❖ ... Separate Chaining

Example of separate chaining ...

h("abc") = 2, h("def") = 1, h("ghi") = 0, h("jkl") = 2, h("mno") =



Concrete data structure for hashing via chaining

```
typedef struct HashTabRep {
   List *lists; // array of Lists of Items
   int  N;      // # elements in array
   int  nitems; // # items stored in HashTable
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->lists = malloc(N*sizeof(List));
   assert(new->lists != NULL);
   for (int i = 0; i < N; i++)
      new->lists[i] = newList();
   new->N = N; new->nitems = 0;
   return new;
}
```

Using the **`List`** ADT, search becomes:

```
#include "List.h"
Item *HashGet(HashTable ht, Key k)
{
   int i = hash(k, ht->N);
   return ListSearch(ht->lists[i], k);
}
```

Even without **`List`** abstraction, easy to implement.

Using sorted lists gives only small performance gain.

Other list operations are also simple:

```
#include "List.h"

void HashInsert(HashTable ht, Item it) {
   Key k = key(it);
   int i = hash(k, ht->N);
   ListInsert(ht->lists[i], it);
}
void HashDelete(HashTable ht, Key k) {
   int i = hash(k, ht->N);
   ListDelete(ht->lists[i], k);
}
```

Essentially: select a list; operate on that list.

Cost analysis:

- N array entries (slots), M stored items
- average list length $L = M/N$
- best case: all lists are same length L
- worst case: one list of length M ($h(k)=0$)
- searching within a list of length n:
  - best: 1, worst: n, average: $n/2 \Rightarrow O(n)$
- if good hash and $M \leq N$, cost is 1
- if good hash and $M > N$, cost is $(M/N)/2$

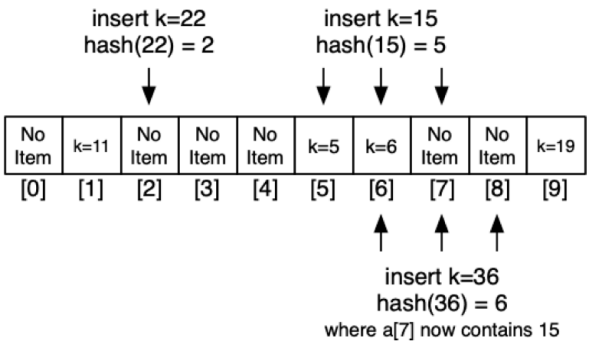
Ratio of items/slots is called **load $\alpha = M/N$**

②. Linear Probing

Collision resolution by finding a new location for **`Item`**

- hash indicates slot *i* which is already used
- try next slot, then next, until we find a free slot
- insert item into available slot

Examples:



How do we find it back?

Search cost analysis:

- cost to reach first **`Item`** is *O(1)*
- subsequent cost depends how much we need to scan
- affected by load α = M/N (i.e. how "full" is the table)
- average cost for successful search = $0.5*(1 + 1/(1-\alpha))$
- average cost for unsuccessful search = $0.5*(1 + 1/(1-\alpha)^2)$

Example costs (assuming large table, e.g. *N>100*):

load (α)	0.50	0.67	0.75	0.90
search hit	1.5	2.0	3.0	5.5
search miss	2.5	5.0	8.5	55.5

Assumes reasonably uniform data and good hash function.

Concrete data structures for hashing via linear probing:

```
typedef struct HashTabRep {
   Item **items; // array of pointers to Items
   int  N;       // # elements in array
   int  nitems;  // # items stored in HashTable
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item *));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++) new->items[i] = NULL;
   new->N = N; new->nitems = 0;
   return new;
}
```

Insert function for linear probing:

```
void HashInsert(HashTable ht, Item it)
{
   assert(ht->nitems < ht->N);
   int N = ht->N;
   Key k = key(it);
   Item **a = ht->items;
   int i = hash(k,N);
   for (int j = 0; j < N; j++) {
      if (a[i] == NULL) break;
      if (equal(k,key(*(a[i])))) break;
      i = (i+1) % N;
   }
   if (a[i] == NULL) ht->nitems++;
   if (a[i] != NULL) free(a[i]);
   a[i] = copy(it);
}
```

Correct.

Searching for next available space.

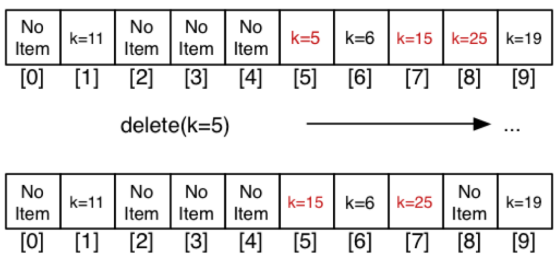
Search function for linear probing:

```
Item *HashGet(HashTable ht, Key k)
{
    int N = ht->N;
    Item **a = ht->items;
    int i = hash(k,N);
    for (int j = 0; j < N; j++) {
        if (a[i] == NULL) break;
        if (equal(k,key(*(a[i]))))
            return a[i];
        i = (i+1) % N;
    }
    return NULL;
}
```

Deletion

Deletion slightly tricky for linear probing.

Need to ensure no **NULL** in middle of "probe path"
(i.e. previously relocated items moved to appropriate location)



(delete & re-insert to see whether that is probing item)

if yes,

Need to move all linear probing item to front, fill up the deleted space.

Delete function for linear probing:

```
void HashDelete(HashTable ht, Key k)
{
   int N = ht->N;
   Item *a = ht->items;
   int i = hash(k,N);
   for (int j = 0; j < N; j++) {
      if (a[i] == NULL) return; // k not in table
      if (equal(k,key(*(a[i])))) break;
      i = (i+1) % N;
   }
   free(a[i]);  a[i] = NULL;  ht->nitems--;
   // clean up probe path
   i = (i+1) % N;
   while (a[i] != NULL) {
      Item it = *(a[i]);
      a[i] = NULL;  // remove 'it'
      ht->nitems--;
      HashInsert(ht, it); // insert 'it' again
      i = (i+1) % N;
   }
}
```

h("ab") = 2, h("cd") = 1, h("ef") = 0, h("gh") = 2, h("ij") = 1

Initially

items[]

[0]	[1]	[2]	[3]	[4]	[5]
-	-	-	-	-	-

After inserting "ab" (h=2)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
-	-	"ab"	-	-	-

After inserting "cd" (h=1)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
-	"cd"	"ab"	-	-	-

After inserting "ef" (h=0)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
"ef"	"cd"	"ab"	-	-	-

After inserting "gh" (h=2)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
"ef"	"cd"	"ab"	"gh"	-	-

After inserting "ij" (h=1)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
"ef"	"cd"	"ab"	"gh"	"ij"	-

After deleting "ab" (h=2)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
"ef"	"cd"	"gh"	"ij"	-	-

After deleting "cd" (h=1)

items[]

[0]	[1]	[2]	[3]	[4]	[5]
"ef"	"ij"	"gh"	-	-	-

A problem with linear probing: clusters

E.g. insert 5, 6, 15, 16, 14, 25, with hash(k) = k%10

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-	1	-	-	4	-	-	-	-	-
-	1	-	-	4	5	-	-	-	-
-	1	-	-	4	5	6	-	-	-
-	1	-	-	4	5	6	15	-	-
-	1	-	-	4	5	6	15	16	-
-	1	-	-	4	5	6	15	16	14
25	1	-	-	4	5	6	15	16	14

Making ~~everything~~ expensive.

searching 25

deleting

❖ Double Hashing

Improvement.

Double hashing improves on linear probing:

- by using an increment which ...
  - is based on a secondary hash of the key
  - ensures that all elements are visited
    (can be ensured by using an increment which is relatively prime to N)
- tends to eliminate clusters ⇒ shorter probe paths

To generate relatively prime

- set table size to prime e.g. N=127
- **`hash2()`** in range [1..N1] where N1 < 127 and prime

Concrete data structures for hashing via double hashing:

```
typedef struct HashTabRep {
   Item **items; // array of pointers to Items
   int  N;       // # elements in array
   int  nitems;  // # items stored in HashTable
   int  nhash2;  // second hash mod
} HashTabRep;

#define hash2(k,N2) (((k)%N2)+1)

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item *));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++)
      new->items[i] = NULL;
   new->N = N; new->nitems = 0;
   new->nhash2 = findSuitablePrime(N);
   return new;
}
```

Search function for double hashing:

```
Item *HashGet(HashTable ht, Key k)
{
   Item **a = ht->items;
   int N = ht->N;
   int i = hash(k,N);
   int incr = hash2(k,ht->nhash2);
   for (int j = 0, j < N; j++) {
      if (a[i] == NULL) break;  // k not found
      if (equal(k,key(*(a[i])))) return a[i];
      i = (i+incr) % N;
   }
   return NULL;
}
```

Insert function for double hashing:

```
void HashInsert(HashTable ht, Item it)
{
   assert(ht->nitems < ht->N); // table full
   Item **a = ht->items;
   Key k = key(it);
   int N = ht->N;
   int i = hash(k,N);
   int incr = hash2(k,ht->nhash2);
   for (int j = 0, j < N; j++) {
      if (a[i] == NULL) break;
      if (equal(k,key(*(a[i])))) break;
      i = (i+incr) % N;
   }
   if (a[i] == NULL) ht->nitems++;
   if (a[i] != NULL) free(a[i]);
   a[i] = copy(it);
}
```

Search cost analysis:

- cost to reach first **`Item`** is *O(1)*
- subsequent cost depends how much we need to scan
- affected by load α = M/N (i.e. how "full" is the table)
- average cost for successful search = $\frac{1}{\alpha} ln(\frac{1}{1-\alpha})$
- average cost for unsuccessful search = $\frac{1}{1-\alpha}$

Costs for double hashing (assuming large table, e.g. *N>100*):

load (α)	0.5	0.67	0.75	0.90
search hit	1.4	1.6	1.8	2.6
search miss	1.5	2.0	3.0	5.5

Can be significantly better than linear probing

- especially if table is heavily loaded

❖ Hashing Summary

Collision resolution approaches:

- chaining: easy to implement, allows α > 1
- linear probing: fast if α ≪ 1, complex deletion
- double hashing: faster than linear probing, esp for α ≅ 1

Only chaining allows α > 1, but performance poor when α ≫ 1

For arrays, once *M* exceeds initial choice of *N*,

- need to expand size of array (*N*)
- problem: hash function relies on *N*,
  so changing array size potentially requires rebuiling whole table
- dynamic hashing methods exist to avoid this

Tries

❖ Tries

A trie ...

- is a data structure for representing a set of strings
  - e.g. all the distinct words in a document, a dictionary etc.
- supports string matching queries in *O(L)* time
  - *L* is the length of the string being searched for

Note: generally assume "string" = character string; could be bit-string

Note: Trie comes from *retrieval*; but pronounced as "try" not "tree"

Each node in a trie ...

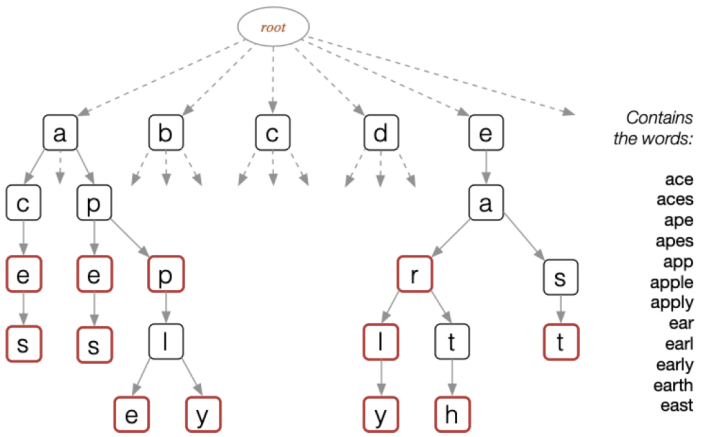
- contains one part of a key (typically one character)
- may have up to 26 children
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children
- may contain other data for application (e.g. word frequency)

A "finishing" node marks the end of one key

- this key may be a prefix of another key stored in trie

Depth *d* of trie = length of longest key value

Trie example:



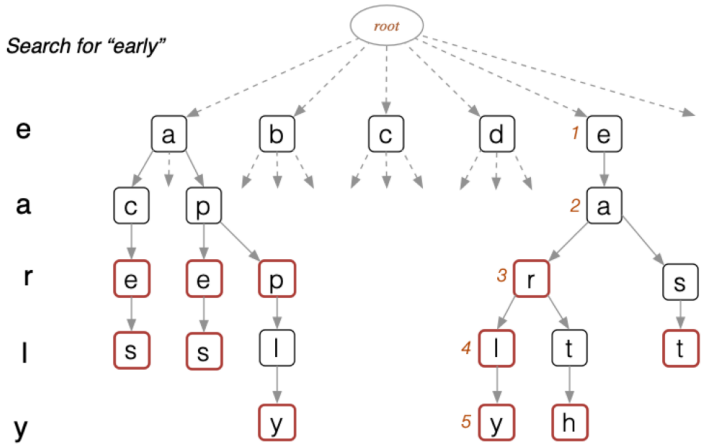
Implementation

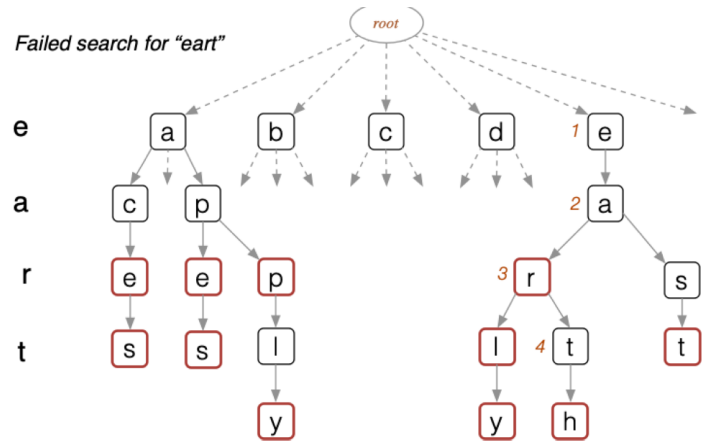
Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
   char onechar;      // current char in key
   Trie child[ALPHABET_SIZE];
   bool finish;       // last char in key?
   Item data;         // no Item if !finish
} Node;

typedef char *Key;    // just lower-case letters
```

Search requires traversing a path, char-by-char from Key:

```
find(trie,key):
|   Input  trie, key
|   Output pointer to element in trie if key found
|          NULL otherwise
|
|   node=trie
|   for each char c in key do
|   |   if node.child[c] exists then
|   |      node=node.child[c]  // move down one level
|   |   else
|   |      return NULL
|   |   end if
|   end for
|   if node.finish then  // "finishing" node reached?
|      return node
|   else
|      return NULL
|   end if
```





Insertion.

Insertion into a Trie ...

```
Trie insert(trie,item,key):
|   Input  trie, item with key of length m
|   Output trie with item inserted
|
|   if trie is empty then
|   |   t=new trie node
|   end if
|   if m=0 then  // end of key
|   |   t.finish=true, t.data=item
|   else
|   |   first=key[0],  rest=key[1..m-1]
|   |   t.child[first]=insert(t.child[first],item,rest)
|   end if
|   return t
```

❖ Cost Analysis

Analysis of standard trie:

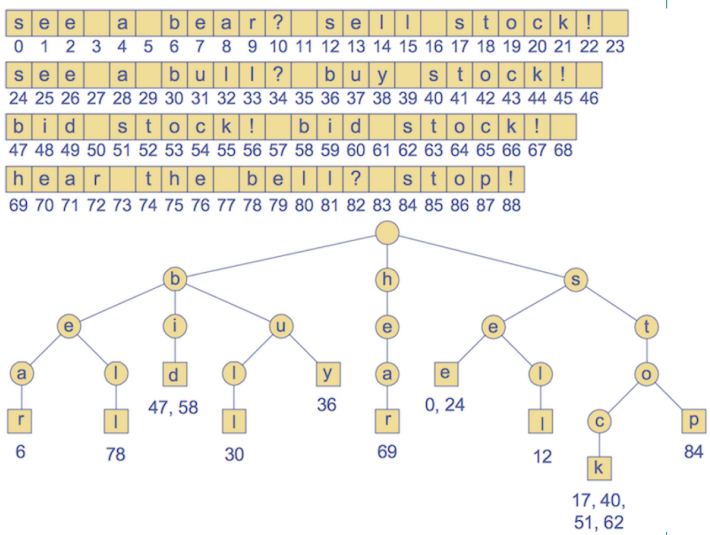
- *O(n)* space
- *O(m)* insertion and search

where

- *n* ... total size of text (e.g. sum of lengths of all strings)
- *m* ... length of the key string
- *d* ... size of the underlying alphabet (e.g. 26)

Example:

Example text and corresponding trie of searchable words:

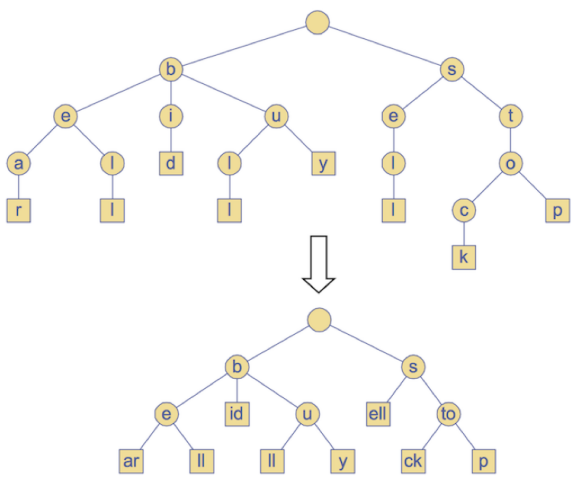


Note: trie has no prefixes ⇒ all finishing nodes are leaves

Compressed tries ...

- have internal nodes of degree ≥ 2; each node contains ≥ 1 char
- obtained by compressing non-branching chains of nodes

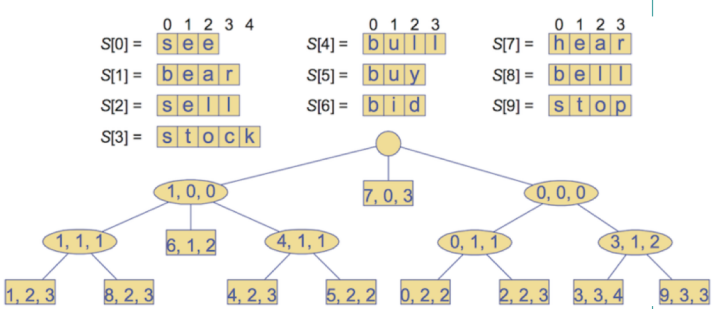
Example:



Compact representation of compressed trie to encode array S of strings:

- nodes store ranges of indices instead of substrings
  - use triple *(i,j,k)* to represent substring *S[i][j..k]*
- requires *O(s)* space (s = #strings in array S)

Example:



Performance Analysis.

Showing which function cost the most time.

❖ gprof: A Profiler

The **gprof** command displays execution profiles ...

- must compile and link program with the **-pg** flag
- executing program creates an new **gmon.out** file
- **gprof** reads **gmon.out** and prints profile on stdout

Example of use:

```
$ gcc -pg -o xyz xyz.c
$ xyz < data > /dev/null
$ gprof xyz | less
```

For further usage details, **man gprof**.

The **gprof** command works at the function level.

It gives a flat profile containing:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

It also gives a call graph, containing:

- which functions called each function
- which functions were called by each function

Consider the following program ...

```
// search for words in text containing a given substring
// display each such word once (in alphabetical order)

int main(int argc, char*argv[])
{
    char  word[MAXWORD];  // current word
    List  matches;        // list of matched words
    char  *substring;     // string to look for
    FILE  *input;         // the input file

    // ... Check command-line args, open input file ...

    // Process the file - find the matching words
    matches = newList();
    while (getWord(input, word) != NULL) {
        if (contains(word,substring)
                && !member(matches,word))
            matches = insert(matches,word);
    }
    printWords(matches);
    return 0;
}
```

Flat profile for this program (**xwords et data3**):

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
75.00	0.03	0.03	30212	0.99	0.99	getWord
25.00	0.04	0.01	30211	0.33	0.33	contains
0.00	0.04	0.00	489	0.00	0.00	member
0.00	0.04	0.00	267	0.00	0.00	insert
0.00	0.04	0.00	1	0.00	40000.00	main
0.00	0.04	0.00	1	0.00	0.00	printWords

The flat profile shows which functions contribute most to the execution cost.

For more info on how to interpret **gprof** flat profiles, look at:

- The GNU Profiler: How to Understand the Flat Profile

Note: **wc data3 → 7439 30211 188259**.

- ~ 75% of the execution time is spent reading words
- ~ 25% of the execution time is spent checking for the substring
- because most words don't contain substring, few calls to **member()**
- there are 30211 words in the input file (+1 **getword()** call to find EOF)
- there are 489 total incl. 267 distinct words containing the substring

Call graph for the same execution (**`xwords et data3`**):

```
index %time    self  children    called     name
                0.00    0.04       1/1           _start [2]
[1]    100.0    0.00    0.04       1         main [1]
                0.03    0.00   30212/30212       getWord [3]
                0.01    0.00   30211/30211       contains [4]
                0.00    0.00     489/489         member [5]
                0.00    0.00     267/267         insert [6]
                0.00    0.00       1/1           printWords [7]
-----------------------------------------------
[2]    100.0    0.00    0.04                 _start [2]
                0.00    0.04       1/1           main [1]
-----------------------------------------------
                0.03    0.00   30212/30212       main [1]
[3]     75.0    0.03    0.00   30212         getWord [3]
-----------------------------------------------
                0.01    0.00   30211/30211       main [1]
[4]     25.0    0.01    0.00   30211         contains [4]
-----------------------------------------------
                0.00    0.00     489/489         main [1]
[5]      0.0    0.00    0.00     489         member [5]
-----------------------------------------------
                0.00    0.00     267/267         main [1]
[6]      0.0    0.00    0.00     267         insert [6]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[7]      0.0    0.00    0.00       1         printWords [7]
```

What does each entry mean?

```
index %time    self  children    called     name
                0.00    0.04       1/1           _start [2]
[1]    100.0    0.00    0.04       1         main [1]
                0.03    0.00   30212/30212       getWord [3]
                0.01    0.00   30211/30211       contains [4]
                0.00    0.00     489/489         member [5]
                0.00    0.00     267/267         insert [6]
                0.00    0.00       1/1           printWords [7]
```

This entry shows info about the **`main()`** function

- **`main()`** is called once, by a **`_start`** "function"
- **`main()`** and its called functions account for 100% of the exection time
- **`main()`** calls five functions (**`getWord`**, **`contains()`**, etc.)
- all calls to the functions come from **`main()`**
  - of the 489 calls to **`member()`**, all of them are made by **`main()`**