Computer Science 2210 – Algorithms

New Finals Material, BUT REVIEW MIDTERM MATERIAL:

(Topic 8 – BST with Dict Map)

Binary search trees (updated) (pdf)

Definitions/Formulas/Methods:

- Dictionary = is an ADT that allows for storage of a collection of records in the form of (key, data). Allowing you to get or remove a key, and put a key, data record inside.

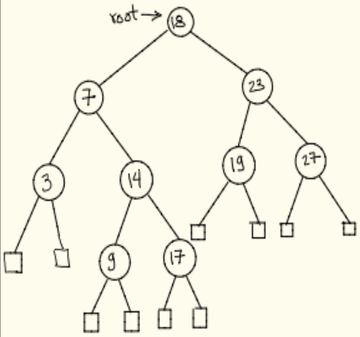
- Search Table = is an ordered map where the items are stored in a key sorted array.
  - O(logn) time for binary search, worst case.
  - O(n) for inserting and removing an item, worst case.
  - Effective for small ordered maps, or maps where searches are rarely done.

Concepts/Formulas:

- An ordered dictionary stores keys based on their values, allowing the use of smallest, largest, predecessor and successor.

- A proper bst stores records so that each key in the left child is smaller than its parent and each key in the right child is greater than its parent. Leaves do not store records, only null.

- An ordered dictionary has a time complexity of O(n) worst case and O(logn) best case, where n = height.



Algorithm smallest(r)

In: Root r of a binary search tree

Out: Node storing smallest key, or null if the tree has no data in it

C_2 { if r is a leaf then return null
else {

p ← r

while p is an internal node do

$C_1 \times (height+1)$ { C_1 { p ← left child of p

return parent of p

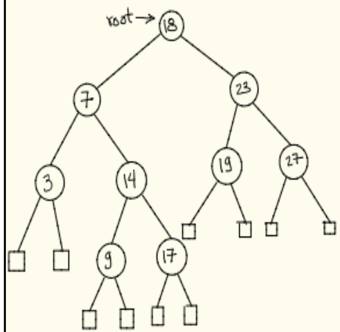
}

iterations = # nodes in leftmost branch

≤ height + 1

$f(n) = \cancel{c_2} + \cancel{c_1}(height + \cancel{1})$

is O(height)



Algorithm get(r,k)

In: Root r of a binary search tree, key k.

Out: Node storing k, or leaf where k should have been stored

C_1 { if r is a leaf then return r

else

if k = key stored in r then

return r

else k < key stored in r then

return $\boxed{\text{get(left child of r, k)}}$

else return $\boxed{\text{get(right child of r, k)}}$

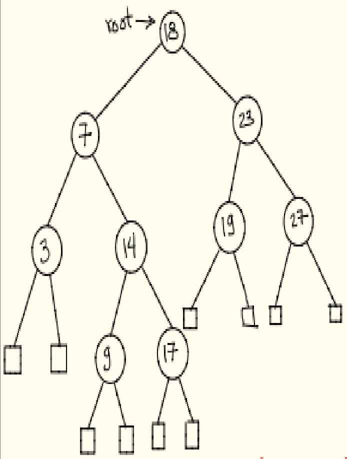
How many calls?

worst case: k not in tree

At most height + 1 calls, so

$f(n) = \cancel{c_1}(height + \cancel{1})$

is O(height)



Algorithm put(r,k,d)

In: Root r of a binary search tree, record (k,d)

Out: True if (k,d) was added to the tree, false otherwise

O(height) { p ← get(r,k)

C_1 { if p is internal node then

return false

else {

p.key ← k

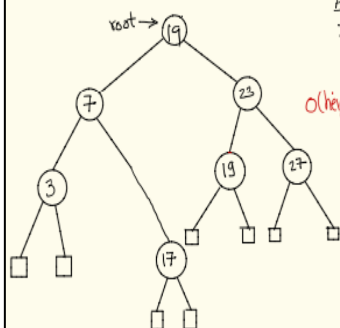
p.data ← d

Create 2 leaves and set them as children of p

return true

}

f(n) is O(height)



Algorithm remove(r,k)

In: Root r of a binary search tree, key k

Out: True if k was removed, false otherwise

O(height) { p ← get(r,k)

C_1 { if p is a leaf then return false

else {

if p has a child c that is a leaf then {

p ← parent of p

c' ← the other child of p

if p is the root then

Make c' the new root

else

Make c' the child of p'

}

else {

O(height) { s ← smallest(right child of p)

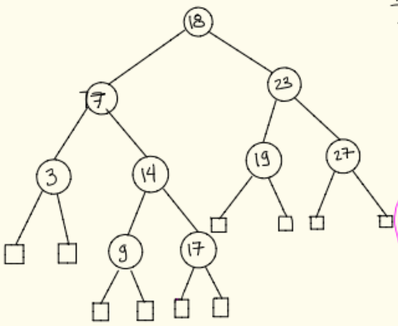
C_2 { Copy key, data from s to p

O(height) { remove(s, key in s)

}

Time complexity

f(n) is O(height)



root → □

Algorithm successor(r, k)

In: Root r of a binary search tree, key k

Out: Node storing the successor of k, or null if k has no successor

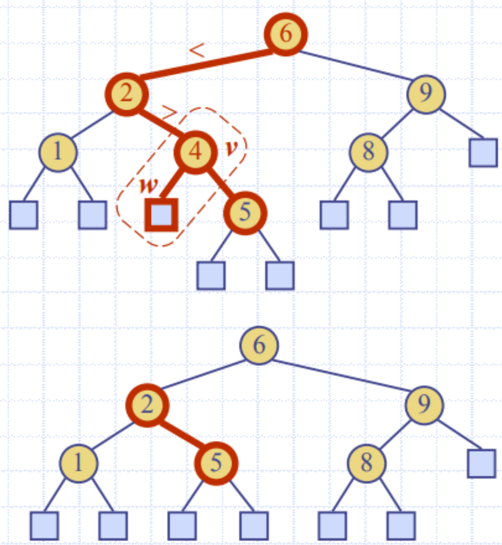
```
if r is a leaf then return null
else {
    p ← get(r,k)  } O(height)
    if (p is an internal node) and (right child of p is internal) then
        return smallest(right child of p)      O(height)
    else {
        p' ← parent of p
        while (p ≠ r) and (p is the right child of p') do {
            p ← p'
            p' ← parent of p
        }                                      c2 × height   c2
        if p=r then return null
        else return p'
    }                                          c1
}
```

f(n) is O(height) = O(log n)

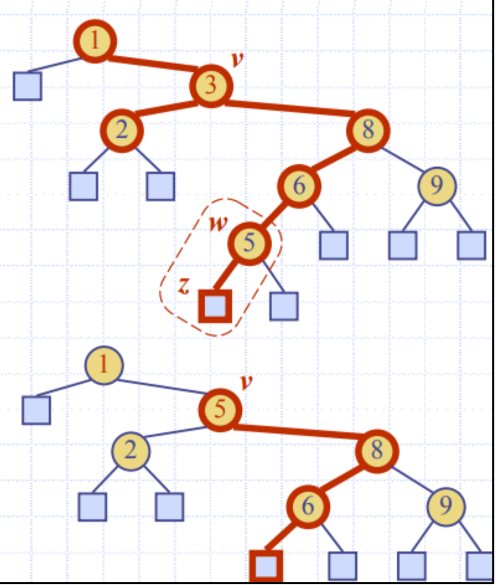
Algorithm *TreeSearch*(*k*, *v*)
  if *v* is a leaf **then**
    return *v*
  if $k < key(v)$ **then**
    return *TreeSearch*(*k*, *left*(*v*))
  else if $k = key(v)$ **then**
    return *v*
  else { $k > key(v)$ }
    return *TreeSearch*(*k*, *right*(*v*))

Deletion

- To perform operation remove(*k*), we search for key *k*
- Assume key *k* is in the tree, and let let *v* be the node storing *k*
- If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation removeExternal(*w*), which removes *w* and its parent
- Example: remove 4



- We consider the case where the key *k* to be removed is stored at a node *v* whose children are both internal
  - we find the internal node *w* that follows *v* in an inorder traversal
  - we copy *key*(*w*) into node *v*
  - we remove node *w* and its left child *z* (which must be a leaf) by means of operation removeExternal(*z*)
- Example: remove 3



Definitions/Formulas/Methods:

- AVL tree = a balanced BST where for every internal node, the height of its children can differ by 1 at most.
- AVL minimum nodes calculator: S(h) = S(h-1) + S(h-2) + 1.

Concepts/Formulas:

- AVL Tree run times: a single rotation is O(1), find/get, insert/put and remove are O(logn).
  - Rebalancing after insert is O(1). And the data structure uses O(n) space.

What is the Maximum Height of an AVL Tree?

Let n(h) = minimum number of nodes in an AVL tree of height h.

$n(0) = 1, n(1) = 3, n(2) = 5, n(3) = 9, n(4) = 15, \ldots$

$n(h) = 1 + n(h-1) + n(h-2) > 2n(h-2)$

Solve the recurrence equation for h even

$n(0) = 1$

$n(h) > 2n(h-2)$

$2n(h-2) > 2^2 n(h-2\times2)$

$2^2 n(h-2\times2) > 2^3 n(h-2\times3)$

...

$2^i n(h-2\times i) > 2^{i+1} n(\underbrace{h-2\times(i+1)}_{=0})$

Then, $n(h) > 2^{i+1} n(0) = 2^{i+1}$

Since $h-2\times(i+1) = 0$, then $i+1 = h/2$ and so

$n(h) = n > 2^{i+1} = 2^{h/2}$

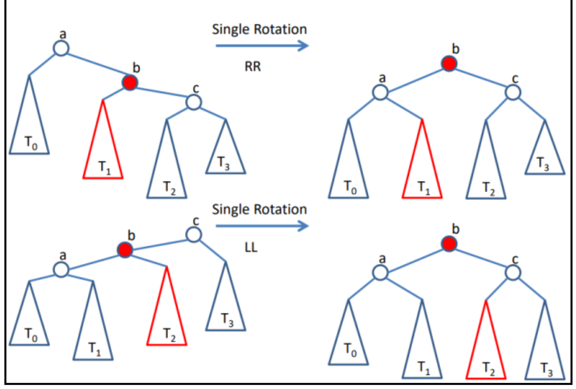
Therefore, taking logarithms on both sides we get

$h/2 \le \log_2 n$

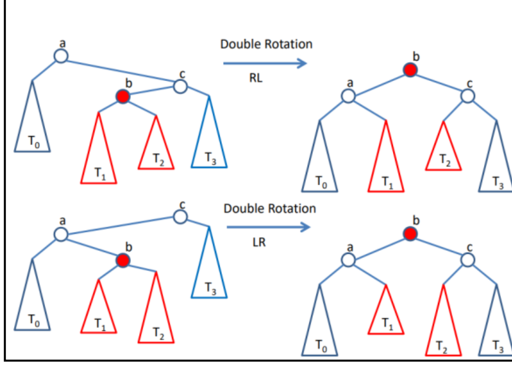
and so

height = $h < 2 \log_2 n$, so height is O(log n)

Single Rotations



Double Rotations

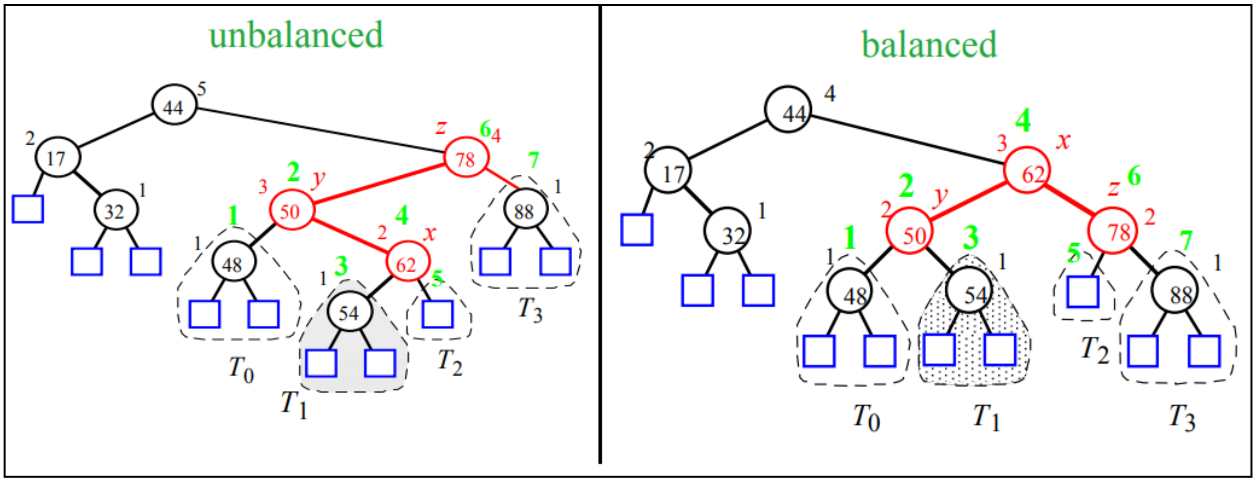


Algorithm putAVL (*r*, *k*, data)
In: Root *r* of an AVL tree, record (*k*,data)
Out: {Insert (*k*,data) and re-balance if needed}

put(*r*,*k*,data)   // Algorithm for binary search trees
Let *p* be the node where (*k*,data) was inserted
while (*p* ≠ null) **and** (subtrees of *p* differ in height ≤ 1) **do**
    p = parent of *p*
if *p* ≠ null **then** rebalance subtree rooted at *p* by
    performing appropriate rotation

Algorithm removeAVL (*r*, *k*)
In: Root *r* of an AVL tree, key *k* to remove
Out: {Remove *k* and re-balance if needed}

remove(*r*,*k*)   // Algorithm for binary search trees
Let *p* be the parent of the node that was removed
while (*p* ≠ null) **do {**
    if subtrees of *p* differ in height > 1 **then**
        rebalance subtree rooted at *p* by performing appropriate rotation
    p = parent of *p*
}



(Topic 10 – Multiway Search Trees)

Multiway Search Trees (pdf)

(2,4)-trees (pdf)

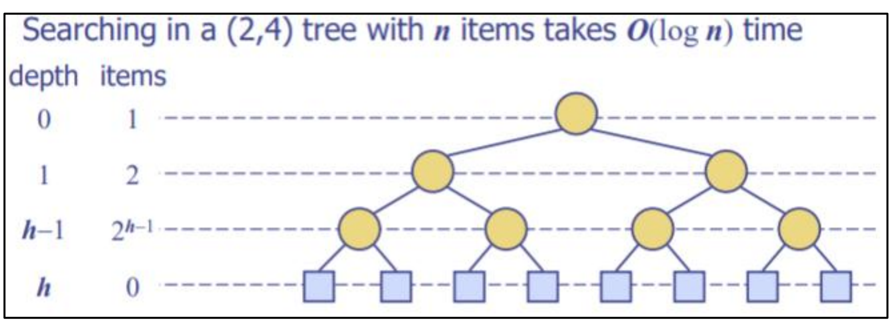
B-trees (pdf)

Definitions/Formulas/Methods:

- Multi-Way Search Tree = an ordered tree where each internal node has at most ‘d’ children and stores ‘d-1’ data items.
  - The first key, k0 will be – infinity and the last key kd will be infinity.
  - Leaves store no items.

- (2,4) Tree = a multi-way search tree where every internal node has 2-4 children, and all the leaves are on the same level.
  - Has a height of O(logn), where n is the amount of items.
  - There are at least 2^i items at depth, so i=0, … h-1. And with no items n => 1 + 2 + 4 + … + 2^{h-1} = 2^h-1
  - → so h <= log (n + 1)

- Order D B-Tree = the root has 2-d children, internal nodes have d/2 – d children and leaves are on the same level.
  - Height is O($\log_d$ n)



Concepts/Formulas:

- Multi Way Tree – get function - time complexity:
  - F(n) = c + (c1 + c2 log d) x tree height
  - → O(log d x tree height)

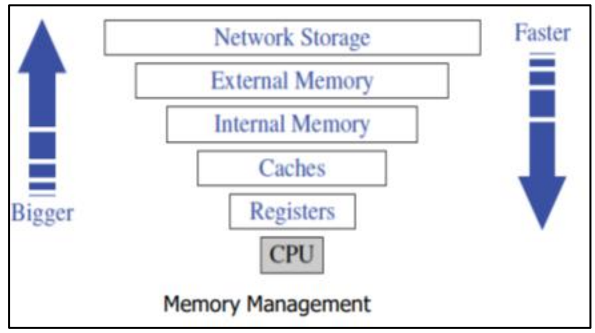
- Multi Way Tree – smallest function - time complexity:
  - F(n) = c x tree height
  - → O(tree height)

- Successor:
  - Get + smallest therefore = get's time complexity.

- Remove:
  - O(d + logd x height)

Multi-Way Searching

Algorithm get(r,k)
In: Root r of a multiway search tree, key k
Out: data for key k or null if k not in tree
if r is a leaf **then return** null
else {
    Use binary search to find the index i such that
        r.keys[i] ≤ k < r.keys[i+1]
    if k = r.keys[i] **then return** r.data[i]
    else return get(r.child[i],k)
}



Algorithm *put* (*r*,*k*,*o*)
In: Root *r* of a (2,4) tree, data item (*k*,*o*)
Out: {Insert data item (*k*,*o*) in (2,4) tree
    Search for ***k*** to find the lowest insertion internal node ***v*** O(log n)
    Add the new data item (***k***, ***o***) at node ***v*** O(1)
    while node *v* ***overflows*** **do** { if a 5 node emerges
        if ***v*** is the root **then**
            Create a new empty root and set as parent of ***v***
        Split ***v*** around the second key ***k'***, move ***k'*** to parent, and update parent's children
        v ← parent of ***v***
    }
O(log n) O(1)
Time complexity of put is O(log n)

Algorithm *remove*(*r*,*k*)
In: Root *r* of a (2,4) tree, key *k*
Out: {remove data item with key *k* from the tree}
    Find the node ***v*** storing key ***k*** O(log n)
    Remove (***k***, ***o***) from ***v*** replacing it with successor if needed
    while node *v* ***underflows*** **do** { if a 1 node emerges O(log n)
        if ***v*** is the root then
            make the first child of ***v*** the new root
        else if a sibling has at least 2 keys **then**
            perform a transfer operation
        else {
            perform a fusion operation
            v ← parent of ***v***
        }
    }
O(log n) O(1)
Time complexity of remove: O(log n)

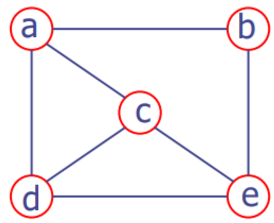
Algorithm *put* (*r*,*k*,*o*)
In: Root *r* of a B-tree, data item (*k*,*o*)
Out: {Insert data item (*k*,*o*) in the B-tree
    Search for ***k*** to find the lowest insertion internal node ***v*** $O(\log d \times \log_d n)$
    Add the new data item (***k***, ***o***) at node ***v*** O(d)
    while node *v* ***overflows*** **do** {
        if ***v*** is the root **then**
            Create a new empty root and set as parent of ***v***
        Split ***v*** around the middle key ***k'***, move ***k'*** to parent, and update parent's children
        v ← parent of ***v***
    }
$O(d \log_d n)$ O(d)
Time complexity of put is $O(d \log_d n)$

Algorithm *remove*(*r*,*k*) Time complexity $O(d \log_d n)$
In: Root *r* of a B-tree, key *k*
Out: {remove data item with key *k* from the tree}
    Find the node ***v*** storing key ***k*** $O(\log d \times \log_d n)$
    Remove (***k***, ***o***) from ***v*** replacing it with successor if needed
    while node *v* ***underflows*** **do** { $O(d + \log d \times \log_d n)$
        if ***v*** is the root then
            make the first child of ***v*** the new root
        else if a sibling has at least ⌈d/2⌉ keys **then**
            perform a transfer operation
        else {
            perform a fusion operation
            v ← parent of ***v***
        }
    }
$O(d \log_d n)$ O(d)

Graphs (pdf)

Definitions/Formulas/Methods:

- Graph = a pair (V, E)
  - V = a set of nodes/vertices.
  - E = a collection of pairs of these nodes, coming together as edges/links.



V = {a,b,c,d,e}
E = {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),(d,e)}

- Complete Graph = has all vertices connected.
- Tree = is a graph without cycles (connecting all the nodes).
- Forest = is a set of trees.

▪ n vertices, m edges	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	$O(\deg(v))$	$O(n)$
areAdjacent (v, w)	$O(m)$	$O(\min\{\deg(v), \deg(w)\})$	$O(1)$
insertVertex(o)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(v, w, o)	$O(1)$	$O(1)$	$O(1)$
removeVertex(v)	$O(m)$	$O(\deg(v))$	$O(n^2)$
removeEdge(v,w)	$O(m)$	$O(\deg(u)+\deg(v))$	$O(1)$

Depth first search (pdf)

Breadth first search (pdf)

Definitions/Formulas/Methods:

- Depth First Search (DFS) =
  - A traversal that visits all the vertices and edges of the graph.
  - Computes the connected components and a spanning forest of the graph.
  - Determines whether the graph is connected.
  - MAIN: It visits children then neighbors. Uses a stack.

- Breadth First Search (BFS) =
  - Does the same as DFS, however it visits neighbors then children. Uses a queue.

BFS Algorithm

```
Algorithm BFS(G, s)
  Q ← new empty queue
  Q.enqueue(s)
  mark(s)
  while Q is not empty do {
    u ← Q.dequeue()
    visit (u)
    for each edge (u,v) incident on u do
        if (u,v) is not labelled then
            if v is not marked then {
                Label (u,v) as DISCOVERY
                mark(v)
                Q.enqueu(v)
            }
            else
                Label (u,v) as CROSS
  }
```

Concepts/Formulas:

- DFS time complexity:
  - Set/get takes O(1).
  - Each vertex and edge is labeled twice.
    1. Initially unexplored
    2. Later visited or (back as an edge).
  - DFS runs in O(n+m) provided the graph is represented by the adjacent list structure
  - DFS runs in $O(n^2)$ if using an adjacency matrix.

```
Algorithm pathDFS(G, v, z)
  Mark(v)
  S.push(v)
  if v = z
    return true
  for all edges (v, w) incident on v do
    if w is not marked then
      if pathDFS(G, w, z) then
        return true
  S.pop(v)
  return false
```

DFS Algorithm from a Vertex

```
Algorithm DFS(u)
In: Vertex u of a graph G
Out: {DFS traversal of G starting at u}
Mark (u)
For each edge (u,v) incident on u do
  if (u,v) is not labelled then
     if v is not marked then {
        Label (u,v) as "discovery edge"
        DFS(v)
     }
     else label (u,v) as "back edge"
```

_____________________________ (Topic 13 – Prim/Djikstra's Alg) _____________________________

Shortest paths, Dijkstra's algorithm (pdf)

Definitions/Formulas/Methods:

- D
- D

Concepts/Formulas:

- d
- d

______________________________ (Topic 14 – Spanning Trees) ______________________________

Minimum spanning trees (pdf).

Definitions/Formulas/Methods:

- D
- D

Concepts/Formulas:

- d
- d