Session 5

Trees - II

Overview

- Balanced Tree
  - AVL Trees
  - Rotations
- Array representation of BST
- Almost Complete Binary Trees
- Heaps and its applications
- External Search
- Multi-way Search Trees
- B-Trees

Balanced Trees

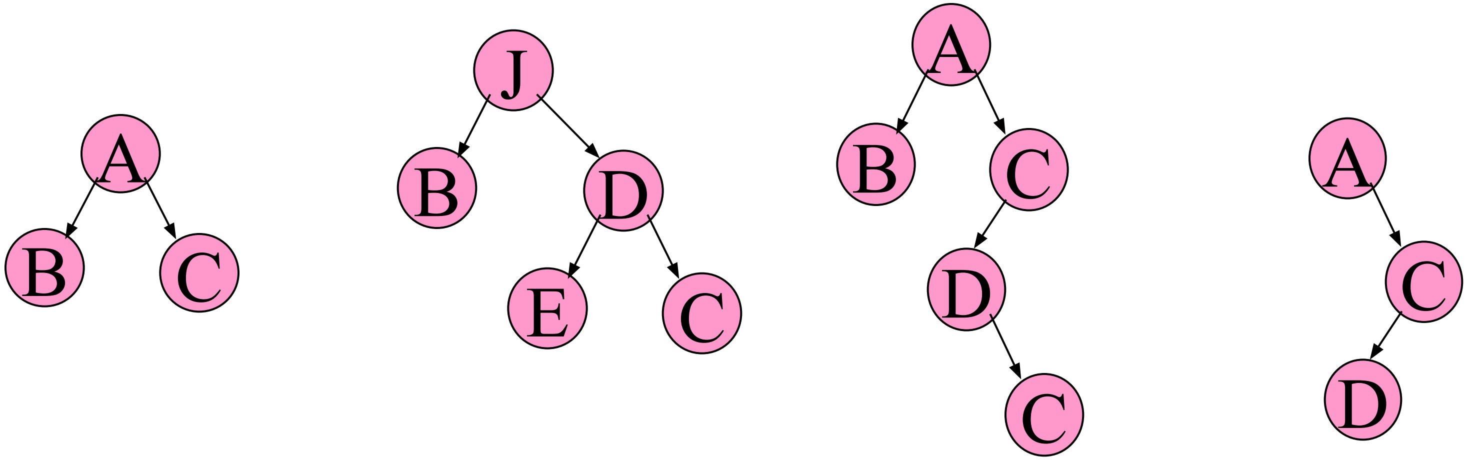
- Depth of average BST is 1.4 times that of completely balanced tree.
- Hence, no major concern about degenerate cases.
- However, over a number of insertions/deletions, tree tends to degenerate.

Balanced Trees

- Also, we want to avoid worst cases - too expensive.
- Hence the interest in keeping the tree reasonably balanced.
- Height Balanced Trees - e.g. AVL trees
- Perfectly balanced trees
  - A height balanced tree where all leaves are at level h(height of the BST) or h –1

AVL Trees

- Height of RST and LST, differ by atmost 1, at all nodes.
- Balance factor = Ht. RST – Ht. LST
- Ensures worst case height of 1.44 log N.
- Thus, about 40% overhead, even in the worst case.

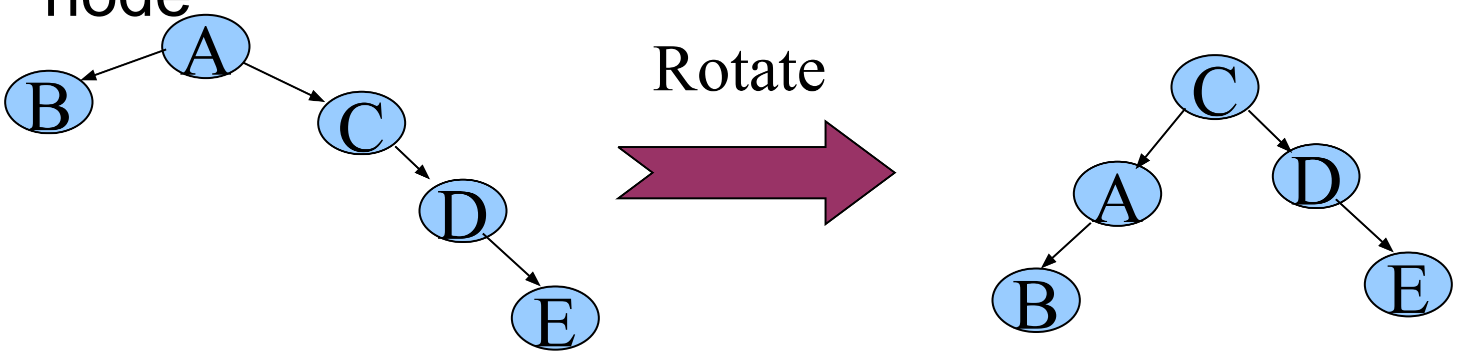


AVL Trees

- Construction done as per BST. When the balance gets violated, some corrective steps are performed.
- Normally, an additional field keeping the balance information is maintained in the nodes.

Rotations

- If balance at a node becomes +2 or -2, AVL criteria at the node is violated.
- Shift the root of the subtree one unit to the heavier side and rearrange the nodes.
- Will make the new balance zero, and total height remains same as the height before the arrival of new node



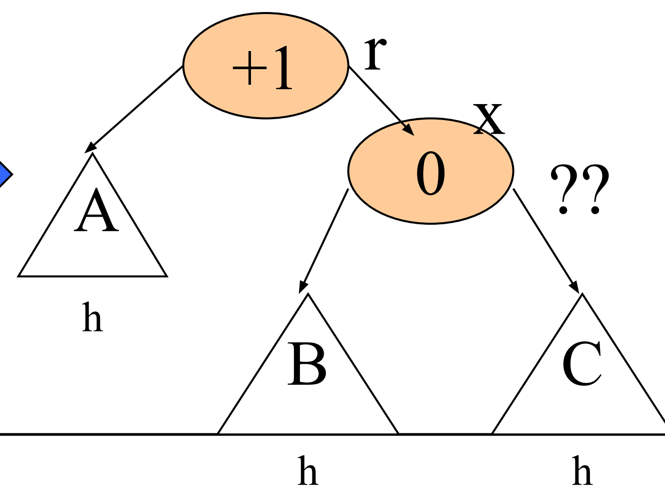
Rotations

- Thus, the parent does not see a height change and hence no balance change
  - Adjustments are confined to a small part of the tree
- Rearranging should also preserve the search tree property.

Restoring Balance

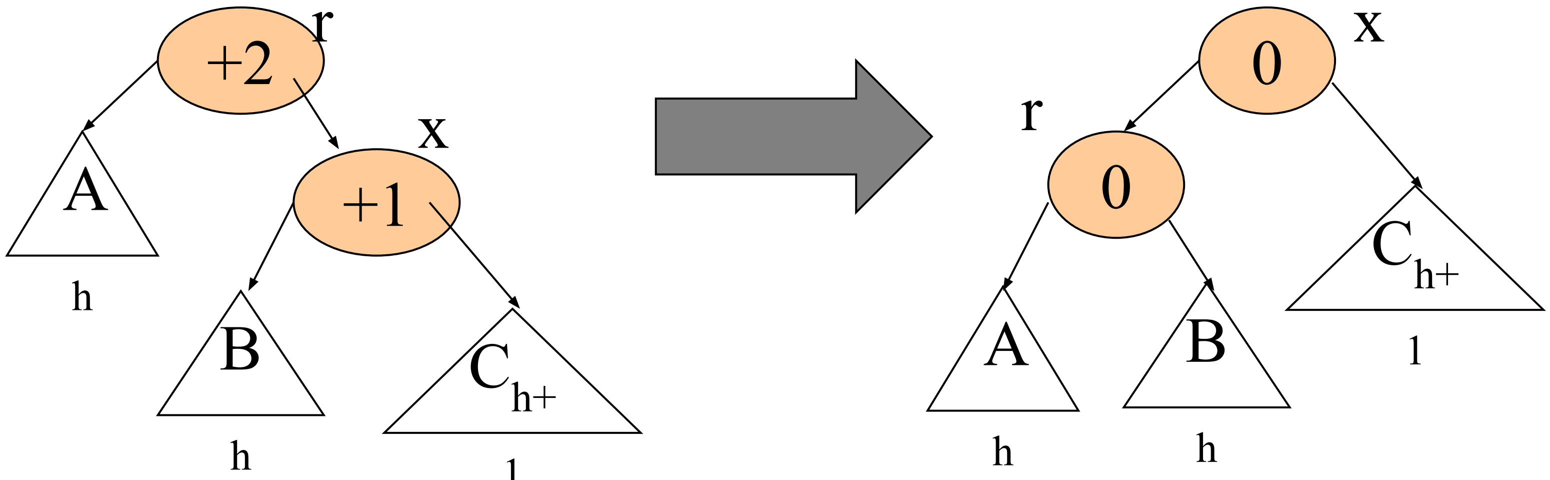
- Consider node r which was at balance +1, with an insertion on the right, making its balance +2.
- r must have had a right child, x with new balance of +1 or -1 (why not zero?).
- Two cases: x becomes +1, x becomes -1

Initial Tree



Restoring Balance: x at +1

- Insertion in C
  - change the root of the subtree to x, make r its LST.

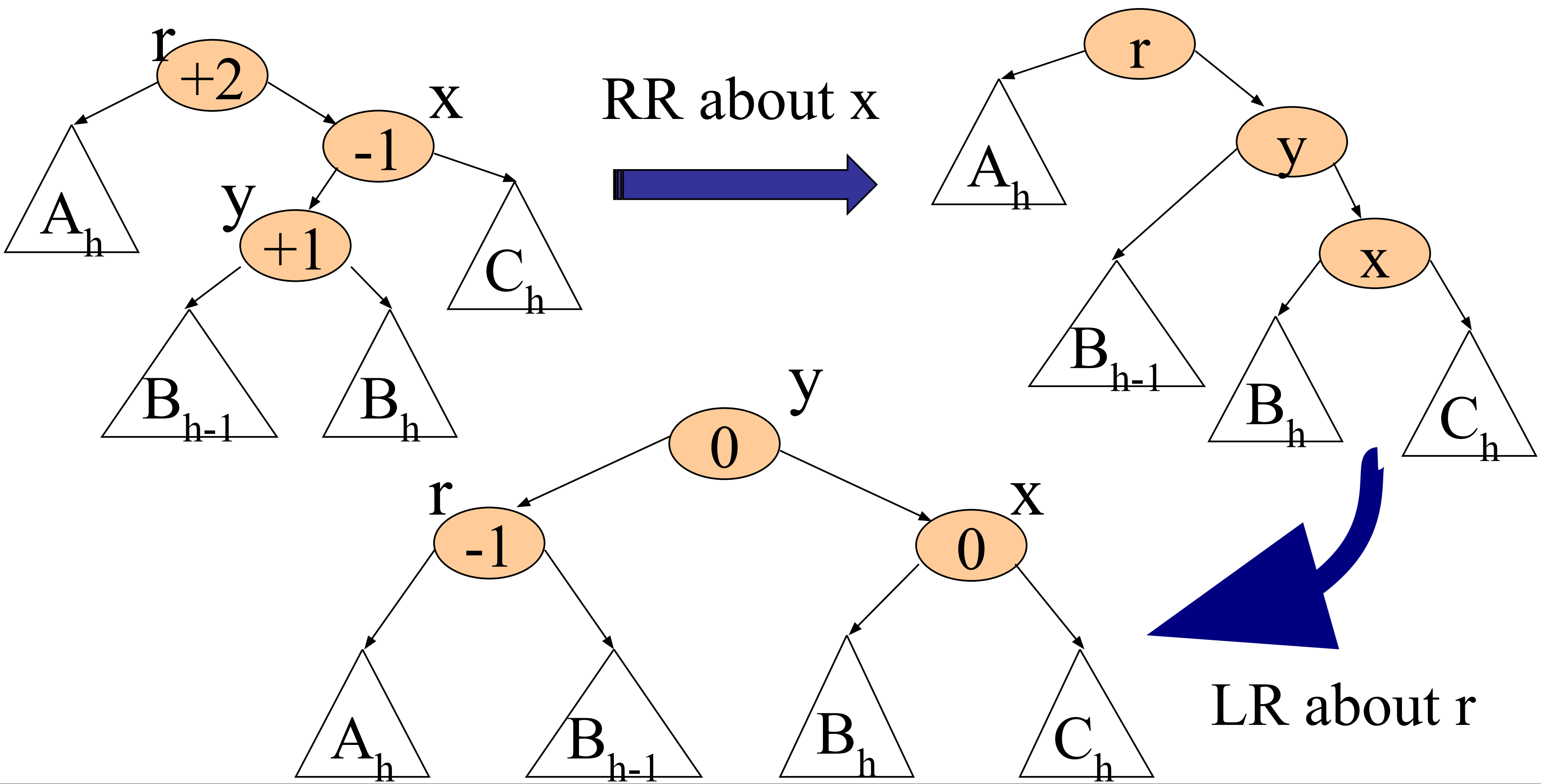


•Height of subtree unaffected (= h+2)

Restoring Balance: x at -1

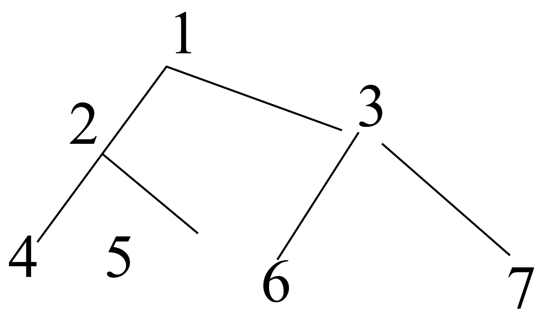
- Simple rotation will not help.
- Examine x's left child, y.
- Perform one rotation on the x-y line to make y as the new root of the RST of r.
- Now RST of r is right heavy.
- Apply the earlier method.

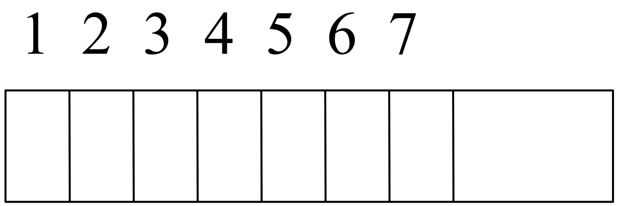
Restoring Balance: x at -1



Array Representation

- Consider a complete binary tree (CBT) and number; its nodes as follows





- View the numbers as indices of an array.
- left-child of node at p will be at 2*p.
- right-child of node at p will be at 2*p+1.
- A node does not have to keep references to its children.

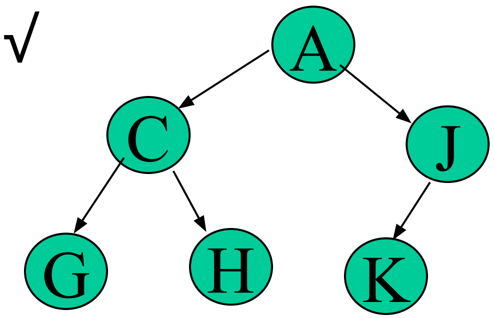
Array Representation

- For arbitrary trees, this results in many vacant spaces.
- Example: a degenerate binary tree of n nodes requires an array of 2^n elements.
- But for CBT or Almost CBT, this can be efficient.
- Also allows us to view a normal array as a binary tree!

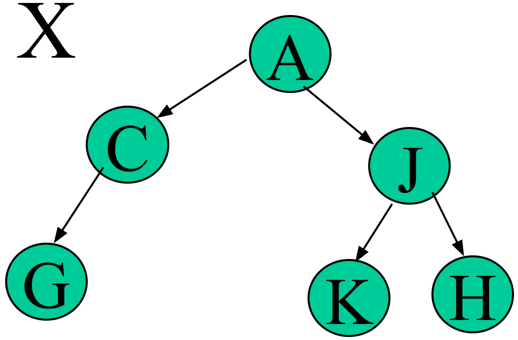
ACBT

- All leafs at lowest and next-to-lowest levels only.
- All except the lowest level is full.
- No gaps except at the end of a level.
- A perfectly balanced tree with leaves at the last level all in the leftmost position.
- A tree which can be represented without any vacant space in the array representation.

√



X



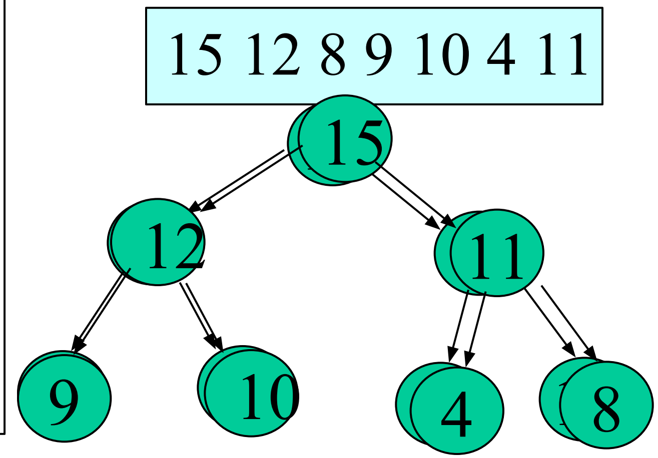
Heaps

- ACBT + a value constraint
- value in node >= value in left-child and >= value in right-child
- No relation between left-child and right-child values.
- No connection with the Heap-memory in Operating Systems.
- Applications
  - Sorting, Priority Queues

Creating a Heap

```
int heap = new int[n];  // n element heap
heap[ii] ≥ heap[2*ii+1], for 0≤ ii ≤(n-1)/2
heap[ii] ≥ heap[2*ii+2], for 0≤ ii ≤(n-2)/2
```

```
heapEnqueue(int el){
Insert el at the end of the heap
while(el is not in the root  and
el > parent(el))
 swap el with parent;
}
```



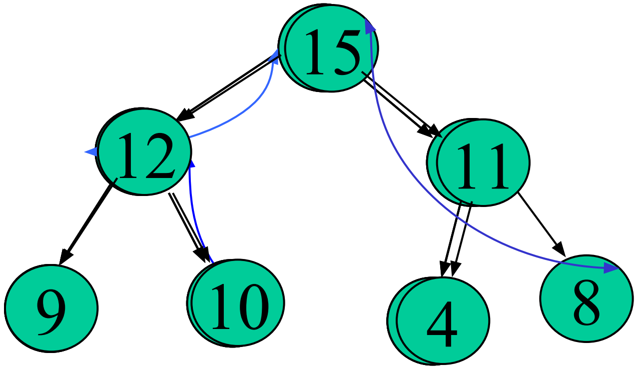
N log(N) complexity

Priority Queue Using Heap

- Descending priority queue: Always select maximum among the available items.
- Heap ensures this at the root.
- After removing the max, readjust the heap: O(log N) algorithm available.
- Insertion can be done as before.

Priority Queue Using Heap

```
heapDequeue()
1.  swap the last element with
    the root element.
2.  P = the root
3.  while(p != leaf &&
        p <any of its children)
         Swap p with larger child;
```



Priority Queue

- PQ with unsorted array gives easy insertion, but retrieval is O(N).
- PQ with sorted array gives easy retrieval, but insertion is O(N).
- PQ with Heap gives both at log N.
- No extra space requirement.

External Search

- Data is partially residing in secondary storage.
- Organizing data stored on disk or tape in an efficient manner.
- Access time about milliseconds, compared to microseconds in memory.
- Comparisons are no more the critical factor, number of disk accesses becomes the focus.

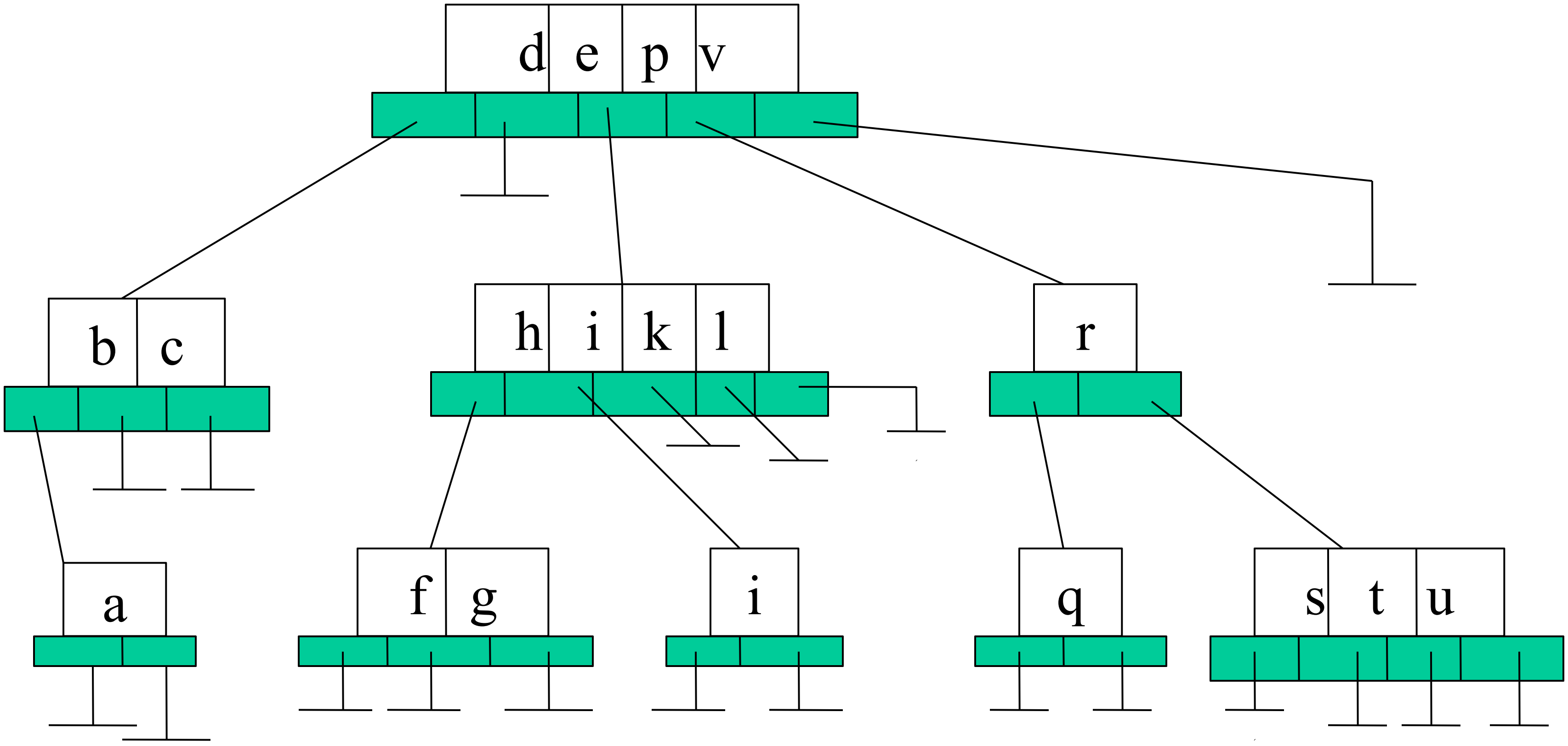
External Search

- Reading one word and one block takes similar time from a disk; therefore, we can pick a larger chunk when reading from disk.
  - increase the node size
  - decrease the height of the tree

Multiway Search Trees

- A node has m-1 keys arranged in order K_1, K_2, …etc.
- Also m children, C_0, C_1, C_2,… etc.
- m is called the Order of the tree.
- When m = 2, we have binary trees (one key, two children).
- Keys and children in a node are ordered as C_0, K_1, C_1, K_2, C_2, ... (all nodes in C_0 have keys less than K_1, etc).

Multiway Search Trees



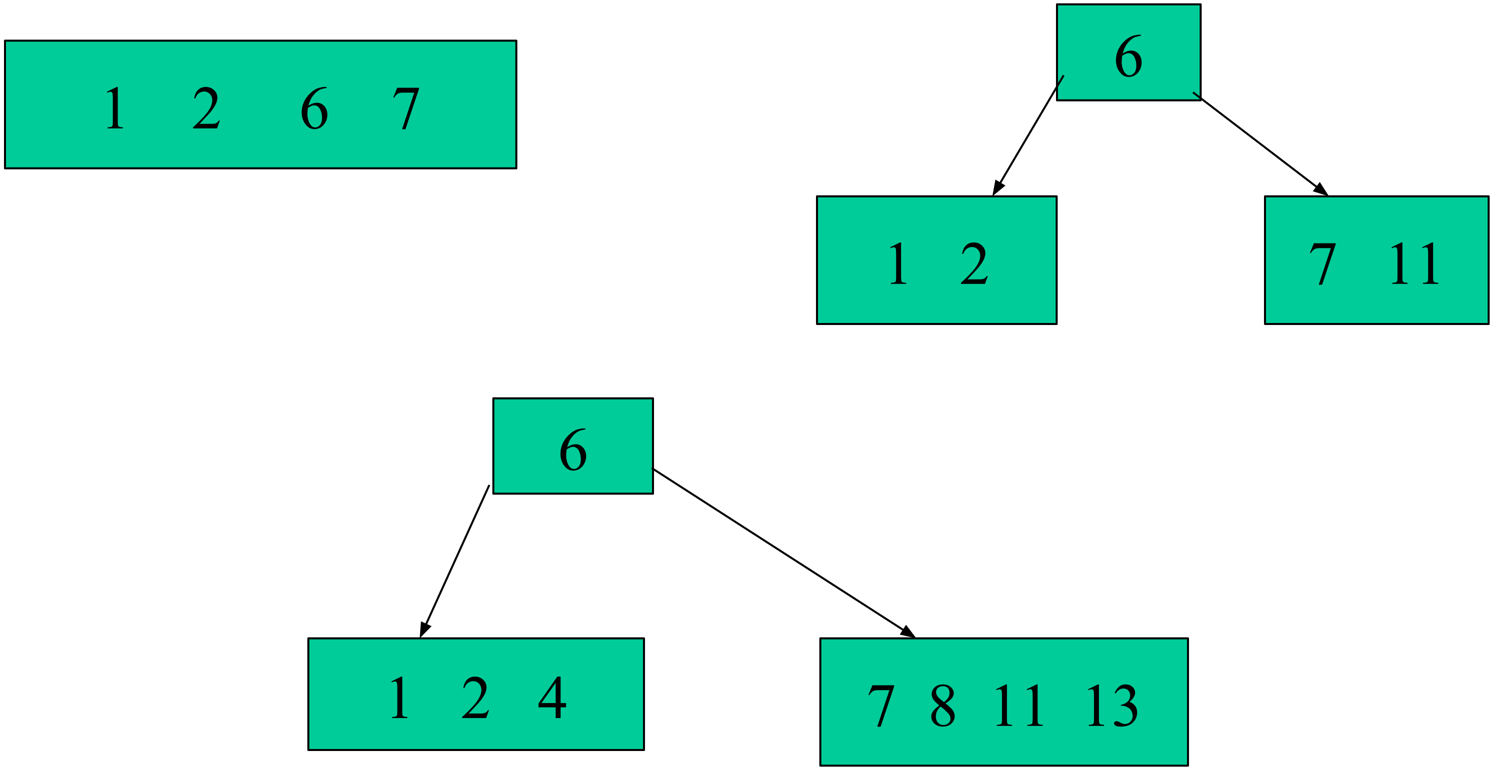
MST … Disadvantages

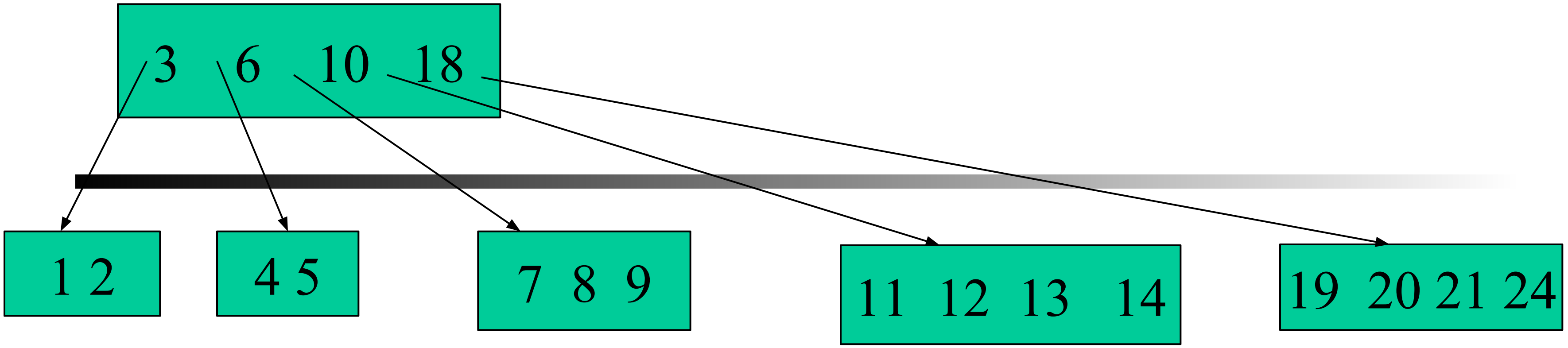
- Like BST, can grow bad cases.
- If most of a node is blank, no advantage over BST.
- Would like to utilize the node capacity properly.
- And also prevent growth of deep trees.
  - B-trees

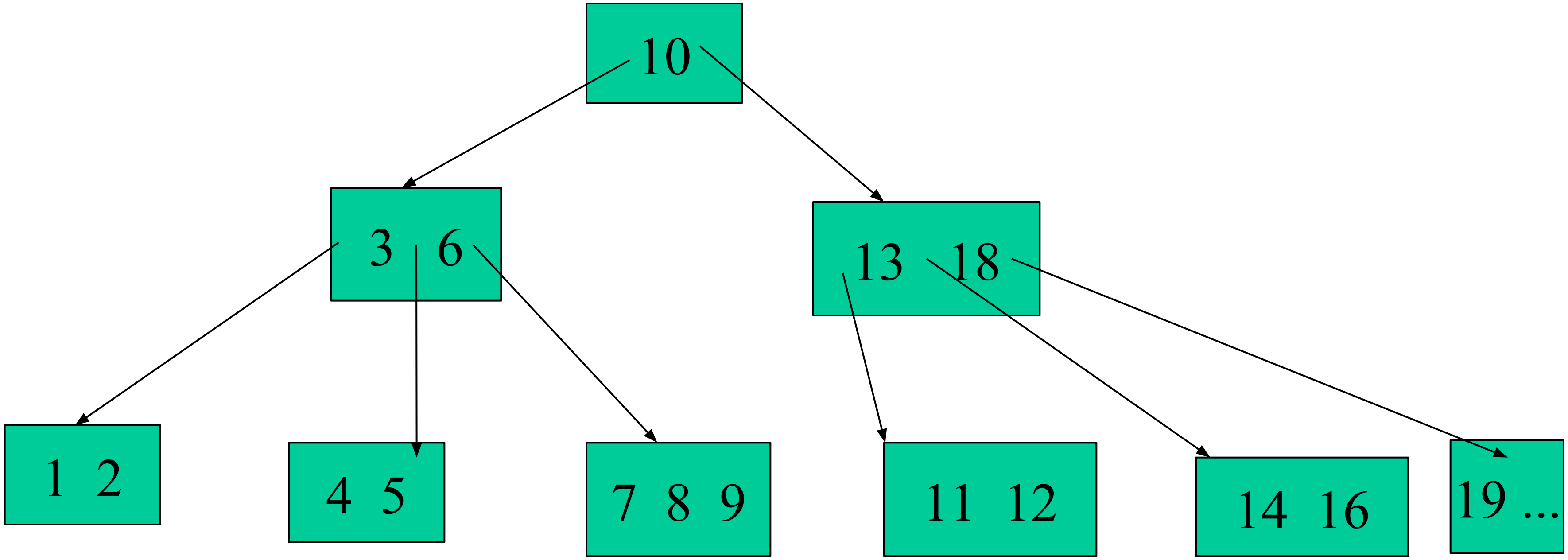
B-Trees

- Balanced MST (order m).
- All leaves at the same level.
- No empty subtree above this level.
- Every node, except the root, has atleast m/2 keys.
- As in MST keys and children in a node are ordered as C_0, K_1, C_1, K_2, C_2, ... (all nodes in C_0 has keys less than K_1 etc).

Insertion sequence: 1, 7, 6, 2, 11, 4, 8, 13, 10, 5, 19, 9, 18, 24, 3, 12, 14, 20, 21, 16




3 6 10 18
1 2
4 5
7 8 9
11 12 13 14
19 20 21 24


10
3 6
13 18
1 2
4 5
7 8 9
11 12
14 16
19 ...

B-Trees … Implementation

```
Class BtreeNode{
  int m = 4; boolean leaf = true;
  int keyTally = 1;
  int keys[] = new int[m-1];
  BtreeNode references[] = new BTreeNode[m];
  BTreeNode(int key){
    keys[0] = key;
    for(int ii=0; ii<m; ii++)
    references[ii] = null;
}}
```

Insertion

- To insert k, traverse from root down, selecting a subtree based on the order.
- If a matching key found, duplicate; no insertion.
- Locate subtree C_i, such that $k < K_i$ and $k > K_{i-1}$ (if $i > 0$).
- If C_i is empty, that is the point of insertion; else repeat.

Insertion

- If the current node (which pointed to C_i) has space, put the key there, rearranging the existing keys.
- Otherwise, split the node into two nodes. The median key value is pulled out, keys less than median form one node, others form the other node.

Insertion

- In the parent, the median key is inserted and the earlier child pointer is replaced by two child pointers for the two new nodes.
- While inserting the key, if the node runs out of space, split that node similarly.
- Repeat the process, till the node which requires no splitting or the current node is the root.

Insertion

- If the root required splitting, create a new root with just the median key and two children pointing to the two fragments of the root so far.
- Since new nodes are created with half-capacity keys, splitting will not be required very frequently.

Implementation … Searching

```
BTreeNode BTreeSearch(int key, BTreeNode node){
 if(node!=null){
   // Search for the left child of the key in the node "node"
  for(int ii=1;ii<node.keyTally &&
                   node.keys[ii-1] < key; ii++)
   if(ii>node.keyTally || node.keys[ii-1] > key)
     return BTreeSearch(key,node.references[ii-1];
   else return node;
  }
 else return null;
}
```

Summary

- Deletion of a node in BST.
- Balanced BST – AVL Trees.
- Rotations at a node in BST.
- Almost complete binary trees and heap.
- External Search.