CS 2302 Data Structures
Fall 2017
B-Trees

1 Definition

A B-tree is a balanced search tree with the following properties:

1. Each node x has the following fields:
   (a) $x.n$ the number of keys stored in x
   (b) $x.key[i]$, $0 <= i < x.n$ the keys
   (c) $x.isLeaf$ a Boolean indicator that is true if x is a leaf
   (d) $x.c[i], 0 <= i <= x.n$, an array of references to the children of x
2. The keys separate the ranges of keys stored in every subtree of x. if k_i is stored in a subtree with root $x.c[i]$, then $k_0 < x.key[0] < k_1 < x.key[1] < x.key[x.n-1] < k_{x.n}$
3. Every leaf has the same depth, which is the tree's height
4. Every B-tree has a parameter t called the *minimum degree*. Every node, with the possible exception of the root, must have at least $t-1$ keys (and t children if it is not a leaf). Every node must have at most $2t-1$ keys (and $2t$ children if it is not a leaf)

1.1 Building a B-tree:

Building a B-tree:

1. Insertions occur at leaves.
2. We start at the root and descend until we find the leaf where the new element should be stored.
3. If a full node is found (one that contains $2t-1$ keys, or, equivalently, where $x.n == x.key.length$) while tracing the path to the appropriate leaf to insert the new element, split it, adding median key to its parent.
4. The height of the tree is increased when we split the root.

As a consequence of the building process, all leaves have the same depth, which is the height of the tree.

1.2 Class definition:

```
public class BTreeNode{
   public int n;           // Actual number of keys on the node
   public boolean isLeaf; // Boolean indicator, true for leaf nodes, false otherwise
   public int[] key;       // Keys stored in the node. They are sorted ion ascending order
   public BTreeNode[] c;  // Children of node. Keys in c[i] are less than key[i] (if it exists)
                          // and greater than key[i-1] if it exists

   public  BTreeNode(int t){  // Build empty node
      isLeaf = true;
      key = new int[2*t-1];   // Array sizes are set to maximum possible value
      c = new BTreeNode[2*t];
      n=0;                    // Number of elements is zero, since node is empty
   }
}
```

2 Problem Solving

We'll cover many different problems using B-trees; however, they can be classified into three types, each involving very similar operations. Understanding how each of them works is crucial to develop your problem-solving skills using this data structure.

2.1 Basic problem types:

Ordered by increasing difficulty, the three problem types are:

1. Traverse a predefined path on the tree. Examples: finding the height of a tree, finding the maximum or minimum key in the tree.
2. Traverse the whole tree. Examples: printing all the keys in the tree, counting the number of keys in the tree, counting the number of nodes in the tree.
3. Searching for a given key. This requires following a path that depends on the given key and the contents of the tree.

2.2 Examples:

Problem type I: finding the smallest key in the tree.

```
int minimum(BtreeNode T){
  if (T.isLeaf)
     return T.key[0];
  return minimum(T.c[0]);
}
```

Problem type II: printing the keys in a B-tree in ascending order.

```
public void printKeys(BtreeNode T){
   if (T.isLeaf){
      for(int i =0; i<T.n;i++)
         System.out.print(T.key[i]+" ");
   }
   else{
      for(int i =0; i<T.n;i++){
         printKeys(T.c[i]);
         System.out.print(T.key[i]+" ");
      }
      printKeys(T.c[T.n]);
   }
}
```

Problem type III: finding the node that contains a given key k given a reference to the root of the tree.

```
BtreeNode SearchBTree(BtreeNode T, int k){
   int i=0;
   while ((i<T.n )&&(k>T.key[i])) //Sequentially search for k
      i++;
   if (i==T.n) || (k<T.key[i])  //k is not in current node.
      if T.isleaf
         return null;
      else
         return SearchBTree(T.c[i], k);
   else
      return T;           //k is in current node.
}
```

2.3 Dealing with depth:

For each of the three types of problems, we may receive an extra parameter or return a value that indicates the depth of the node(s) we are interested in.

2.3.1 Examples:

Problem type I: finding the largest key in the tree at a given depth d. If the height of the tree is less than d return Integer.MIN_VALUE.

```
int maximumAtDepthD(BtreeNode T, int d){
   if (d==0)
       return T.key[T.n-1];
```

```
   if (T.isLeaf)
       return Integer.MIN_VALUE;
   return maximumAtDepthD(T.c[T.n],d-1);
}
```

Problem type II: printing the keys at depth d in a B-tree in ascending order.

```
public void printKeys(BtreeNode T, int d){
   if (d==0)
      for(int i =0; i<T.n;i++)
         System.out.print(T.key[i]+" ");
   else
      if(!T.isLeaf)
         for(int i =0; i<=T.n;i++)
            printKeys(T.c[i],d-1);
}
```

Problem type III: return the depth at which a given key k is found or -1 if k is not in the tree.

```
int SearchBTree(BtreeNode T, int k){
  int i=0;
  while ((i<T.n )&&(k>T.key[i]))//Sequentially search for k
     i++;
  if (i==T.n) || (k<T.key[i])  // k is not in current node.
     if (T.isleaf)             // k is not in the tree
        return -1;
     else{
        int d = SearchBTree(T.c[i], k);
        if (d==-1)             // k is not in the sub-tree
           return -1;
        else
           return d+1;
     }
  else
     return 0;                 //k is in current node.
}
```