B-Trees

1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees
2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

MCS 360 Lecture 35
Introduction to Data Structures
Jan Verschelde, 17 November 2017

B-Trees

balancing search trees

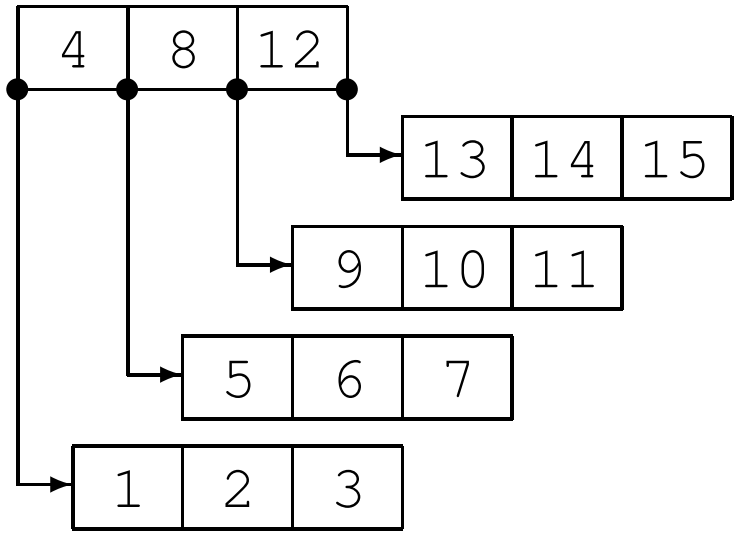
We start with some terminology and an overview:

- A 2-node has two children. A 3-node has three children.
  A 4-node has four children.
- A binary tree is also called a 2-tree as all its nodes are 2-nodes.
  The nodes in a 2-3 tree are 2-nodes or 3-nodes.
  The nodes in a 2-3-4 tree are 2-nodes, 3-nodes, or 4-nodes.
  Consider the red-black tree as a 2-3-4 tree in binary format.
- Balance is maintained in a 2-3 tree and 2-3-4 tree as the insert,
  instead of hanging a new node *onto* a leaf,
  inserts a new node *into* a leaf.
- A B-tree allows for up to n children per node, $n > 2$.
  Applications for B-trees are indices to databases.

As the 2-3-4 tree is a special case of a B-tree, we focus on B-trees.

an example of storing ordered data

A tree with all its 4-nodes filled:

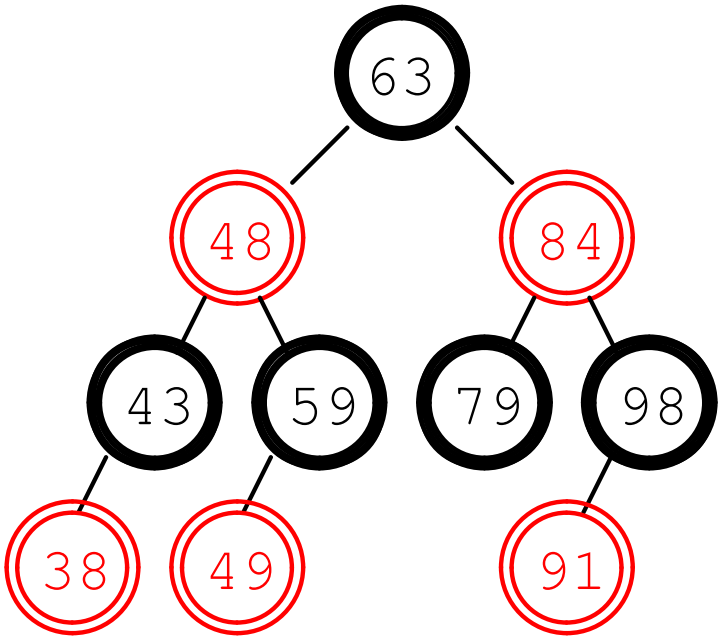


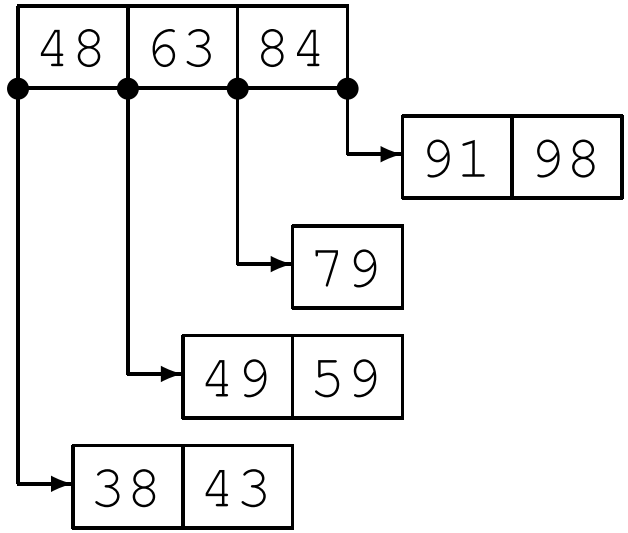
A B-tree is a search tree:

1. data at the nodes are sorted, so we may use binary search;
2. every data element has a left and a right child;
3. at left: less than data element, at right: larger.

relation with red-black trees

The equivalent 2-3-4 tree to the red-black tree is





- A node with no red children is a 2-node.
- A node with one red child is a 3-node.
- A node with two red children is a 4-node.

B-Trees

1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees
2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

node

```
#ifndef MCS360_BTREE_NODE_H
#define MCS360_BTREE_NODE_H
#define NULL 0

struct Node
{
   Item_Type data[numbchil-1];
   // data stored at node
   int size;
   // number of data items, 0 <= size < numbchil-1

   Node* child[numbchil]; // pointers to children

   // for all i: 0 <= i < size :
   //   d in subtree child[i] : d < data[i]
   // and
   //   d > data[size-2], for d in child[size-1].
```

a constructor

```
    Node()  // constructor of node with no data
    {
        size = 0;
        for(int i=0; i<numbchil; i++)
        {
            if(i < numbchil-1) data[i] = Item_Type();
            child[i] = NULL;
        }
    }
};

#endif
```

testing the node

```
#define Item_Type int
#define numbchil 5
#include "mcs360_btree_node.h"

int main()
{
   Node nd;

   for(int i=0; i<numbchil-1; i++)
      nd.data[i] = i+1;

   cout << "Data at the node :";
   for(int i=0; i<numbchil-1; i++)
      cout << " " << nd.data[i];
   cout << endl;

   return 0;
}
```

B-Trees

1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees
2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

a class B_Tree

```
namespace mcs360_btree
{
   template<typename Item_Type, int numbchil>
   class B_Tree
   {
      private:
         #include "mcs360_btree_node.h"
         Node *root;  // data member
      public:
         B_Tree();
         B_Tree(int n, Item_Type *a);
         // creates a tree with n items in a
         // precondition: n < numbchil
         B_Tree(int n, Item_Type *a,
                B_Tree<Item_Type,numbchil> *c);
         // creates a tree with n items in a
         // and n+1 children in c
         // precondition: n < numbchil
```

in mcs360_btree.tc

```
template < typename Item_Type, int numbchil >
B_Tree<Item_Type,numbchil>::B_Tree
  ( int n, Item_Type *a,
    B_Tree<Item_Type,numbchil> *c )
{
   root = new Node;
   root->size = n;
   for(int i=0; i<n; i++)
   {
      root->data[i] = a[i];
      root->child[i] = c[i].root;
   }
   root->child[n] = c[n].root;
}
```

selectors for data

```
template < typename Item_Type, int numbchil >
int B_Tree<Item_Type,numbchil>::get_size()
{
   return root->size;
}

template < typename Item_Type, int numbchil >
Item_Type* B_Tree<Item_Type,numbchil>::get_data()
{
   return root->data;
}
```

selectors for children

```
template < typename Item_Type, int numbchil >
bool B_Tree<Item_Type,numbchil>
   ::is_null_child(int k) const
{
   return (root->child[k] == NULL);
}

template < typename Item_Type, int numbchil >
B_Tree<Item_Type,numbchil>
   B_Tree<Item_Type,numbchil>::get_child(int k)
   const
{
   B_Tree<Item_Type,numbchil> b;
   b.root = root->child[k];
   return b;
}
```

binary_search prototype

```
private:
   int binary_search
     (int first, int last, Item_Type i);

   // Applies binary search to find i in the array
   // starting at first and ending at last.

   // The index on return is either where i occurs,
   // or the child where i should be inserted.

public:
   int search ( Item_Type i );

   // returns the index where i occurs,
   // otherwise returns the child where i
   // could be inserted
```

binary_search definition

```
template < typename Item_Type, int numbchil >
int B_Tree<Item_Type,numbchil>::binary_search
    (int first, int last, Item_Type i)
{
    if(first == last)
        return first;
    else
    {
        int middle = (first + last)/2;

        if(i == root->data[middle])
            return middle;
        else if(i < root->data[middle])
            return binary_search(first,middle,i);
        else
            return binary_search(middle+1,last,i);
    }
}
```

the definition of search

```
template < typename Item_Type, int numbchil >
int B_Tree<Item_Type,numbchil>::search ( Item_Type i )
{
   int L = root->size-1;
   if(i > root->data[L])
      return L+1;
   else
      return this->binary_search(0,L,i);
}
```

writing trees

```
void write_data ( B_Tree<int,5> T )
{
   int *d = T.get_data();

   for(int i=0; i<T.get_size(); i++)
      cout << " " << d[i];
   cout << endl;
}

void write ( int k, B_Tree<int,5> T )
{
   for(int i=0; i<k; i++) cout << "   ";
   write_data(T);
   for(int i=0; i<T.get_size()+1; i++)
      if(!T.is_null_child(i))
         write(k+1,T.get_child(i));
}
```

B-Trees

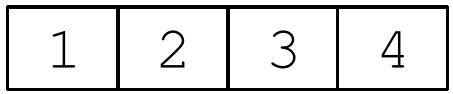
1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees

2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

an example

nodes have 5 children

Inserting 1, 2, 3, 4:



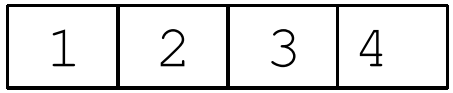
At the start we insert at the root,
and maintain the inserted elements in order.

If the node is full, we split the node at the middle.

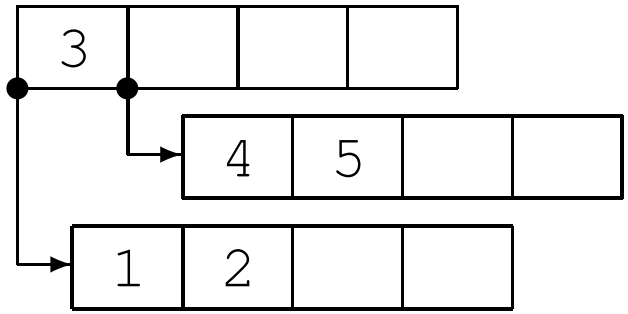
an example

nodes have 5 children

Inserting 5 into:



caused a split in the middle, 3 is new parent:

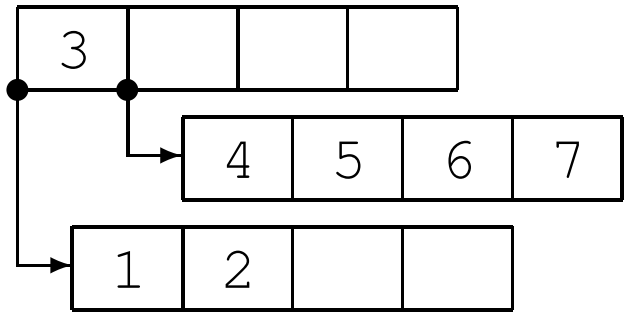


Next we insert 6 and 7 . . .

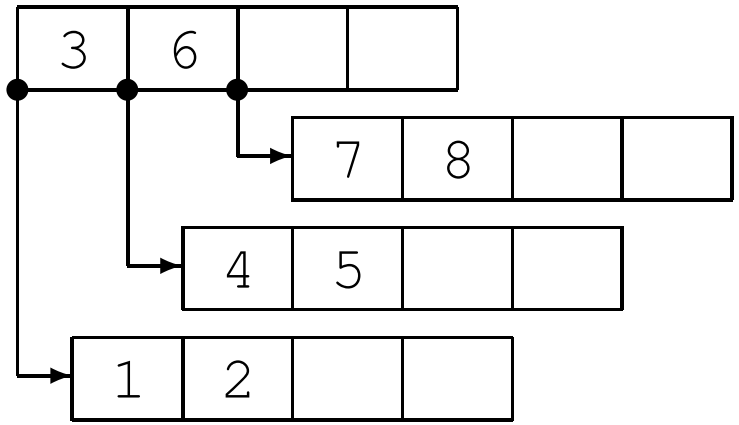
an example

nodes have 5 children

After inserting 6 and 7, if we insert 8 into:



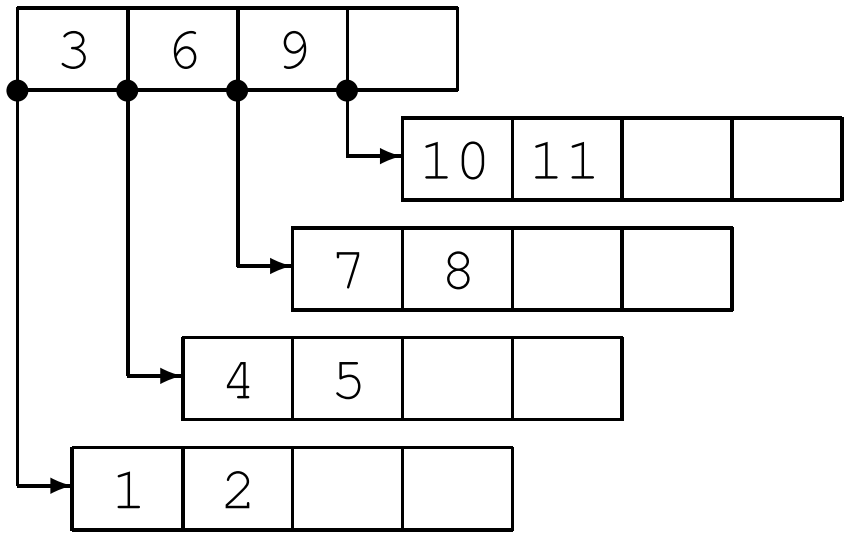
then we cause a split, with 6 as new parent:



an example

nodes have 5 children

After inserting 9, 10, and 11:

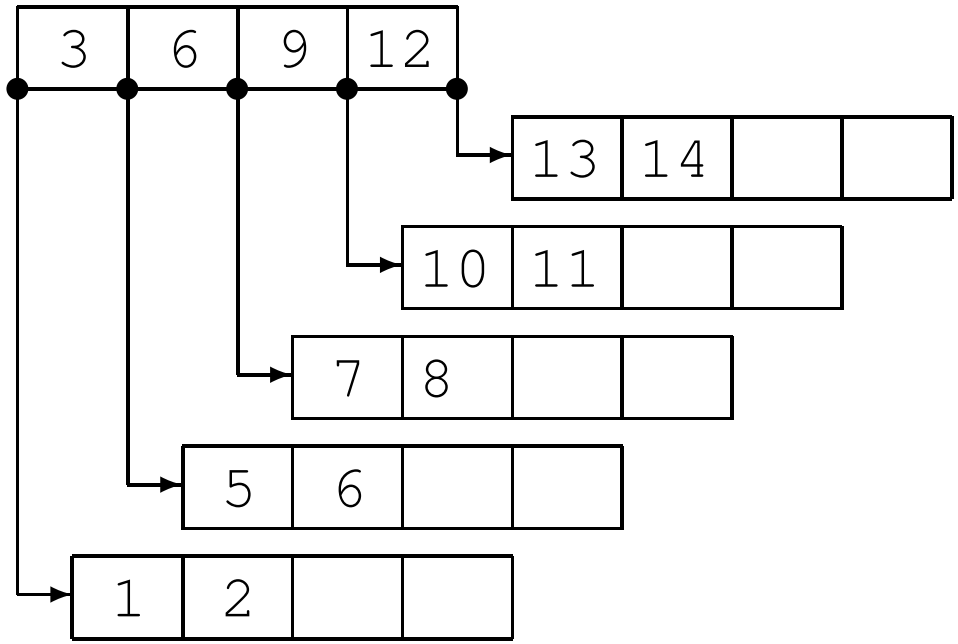


Then we insert 12, 13, and 14 . . .

an example

nodes have 5 children

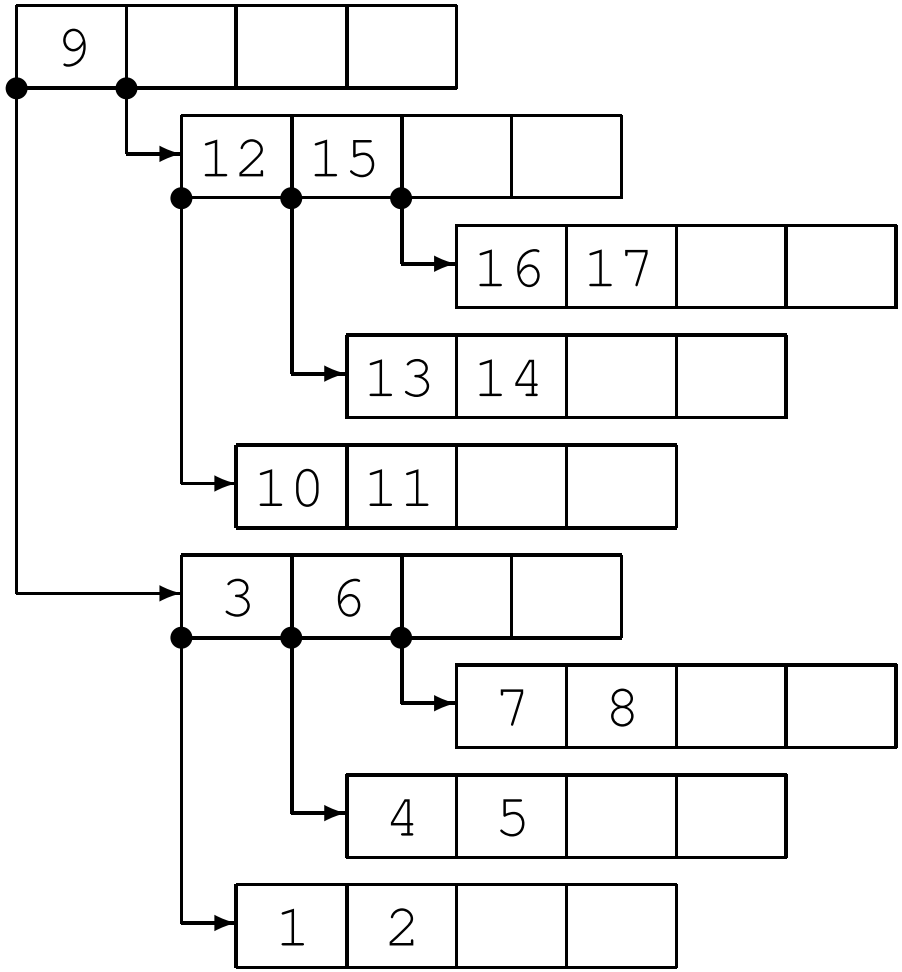
After inserting 12, 13, and 14:



We continue, inserting 15, 16, and 17 . . .

an example

nodes have 5 children



B-Trees

1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees

2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

the insertion algorithm

A B-tree maintains its balance by

- splitting full nodes in half,
- inserting middle element into the parent.

Recursive algorithm to insert:

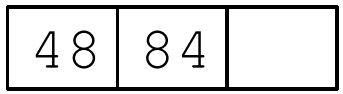
1. find node where to insert,
   decide if simple insert or if split is needed
2. perform simple insert or split and return results
3. after returning from insert, check if split
   place new child and insert item moved up to parent

inserting a sequence of numbers

Insert 48, 84, 43, 91, 38, 79, 63, 59, 49, 98 into a B-tree with $n = 4$.

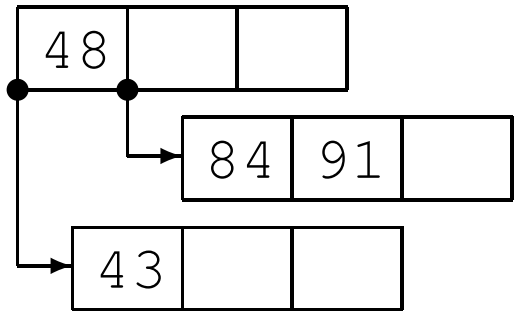
After inserting 48:

After inserting 84:

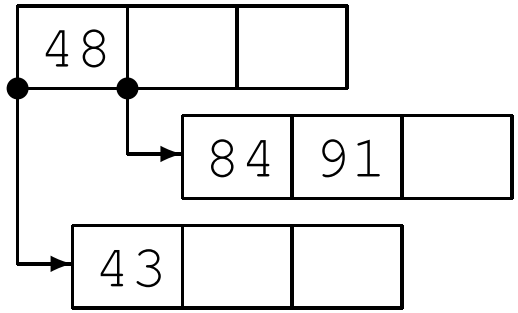


After inserting 43:

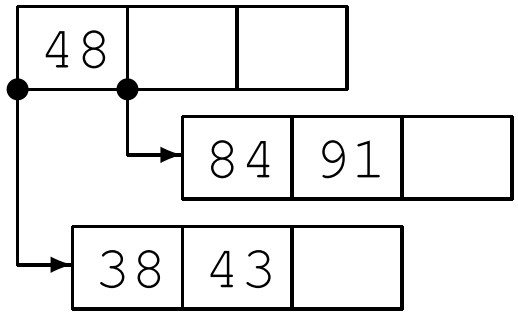
The insert of 91 causes a split in the middle, 48 is the new root:



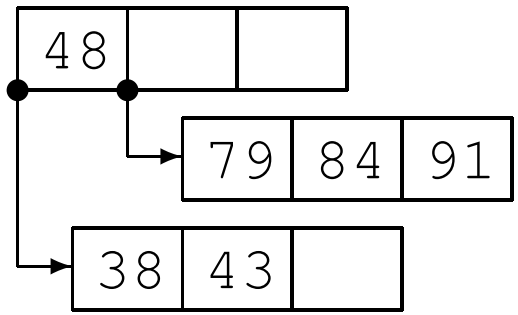
inserting 38, 79, 63, 59, 49, 98



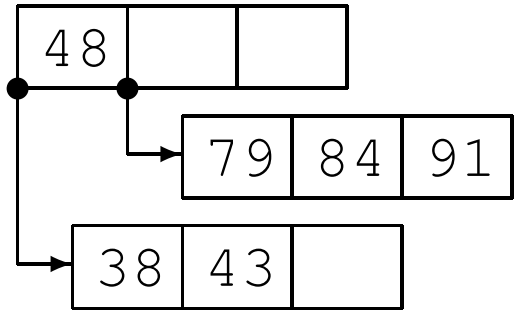
After inserting 38:



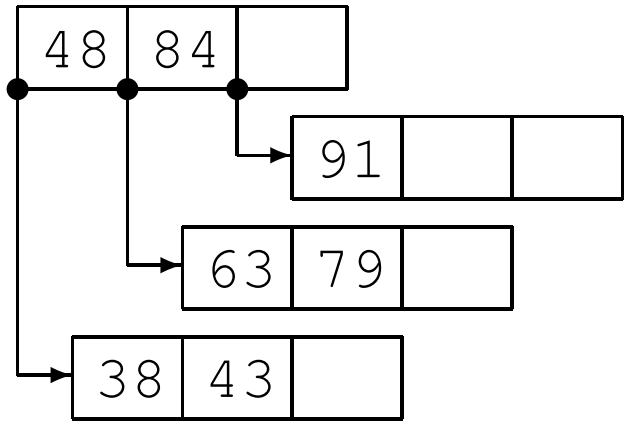
After inserting 79:



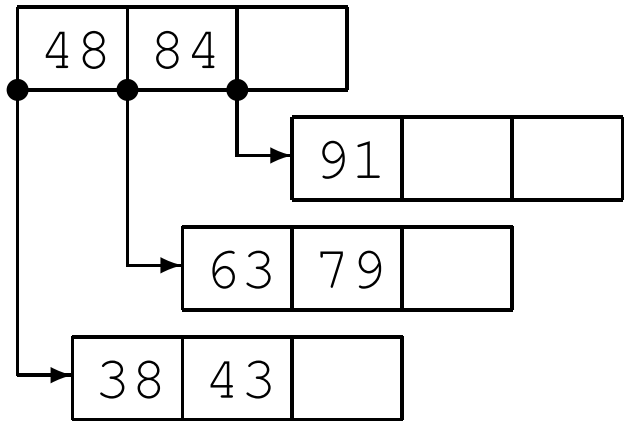
inserting 63 causes a split



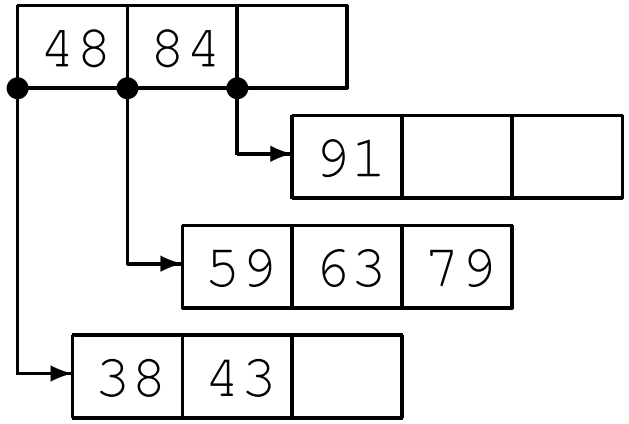
The split leads to an insert of 84 into the root node.
After inserting 63:



inserting 59, 49

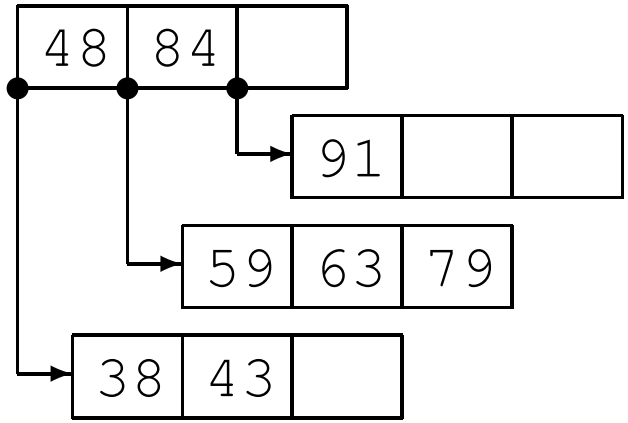


After inserting 59:

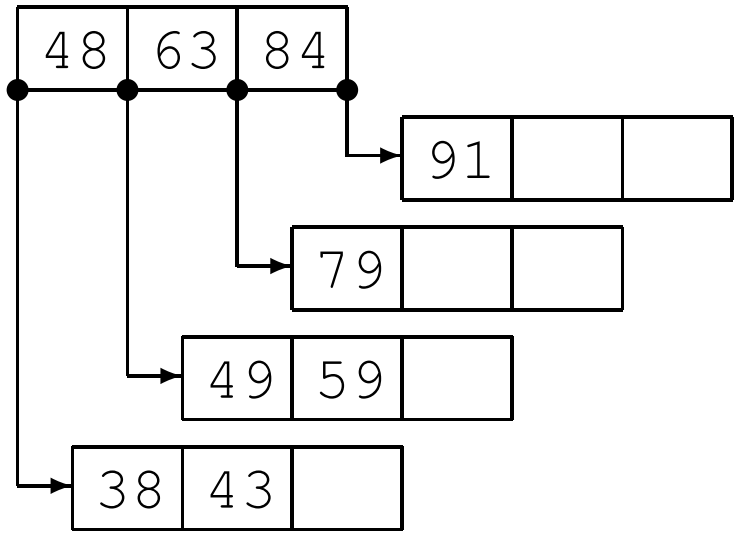


Inserting 49 will cause a split.

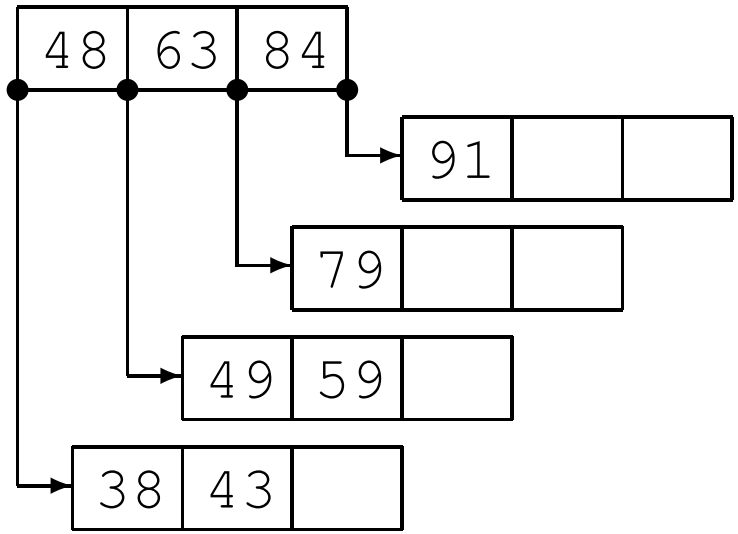
inserting 49 causes a split



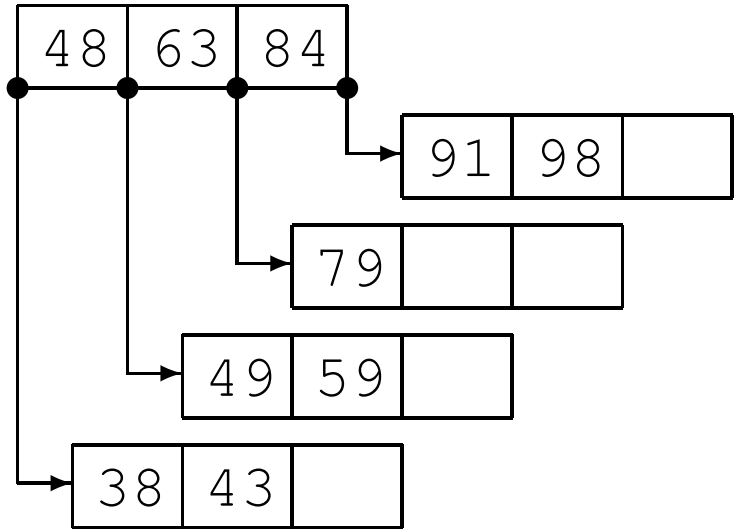
The insert of 49 causes the insert of 63 into the root node:



finally, we insert 98

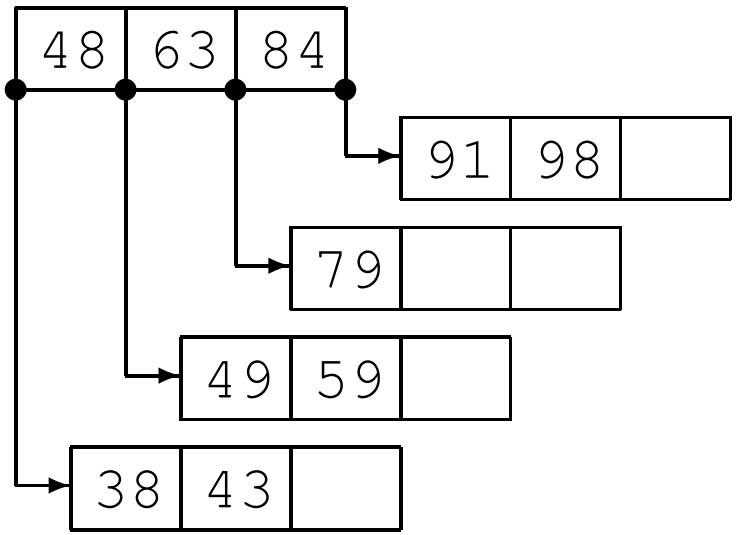


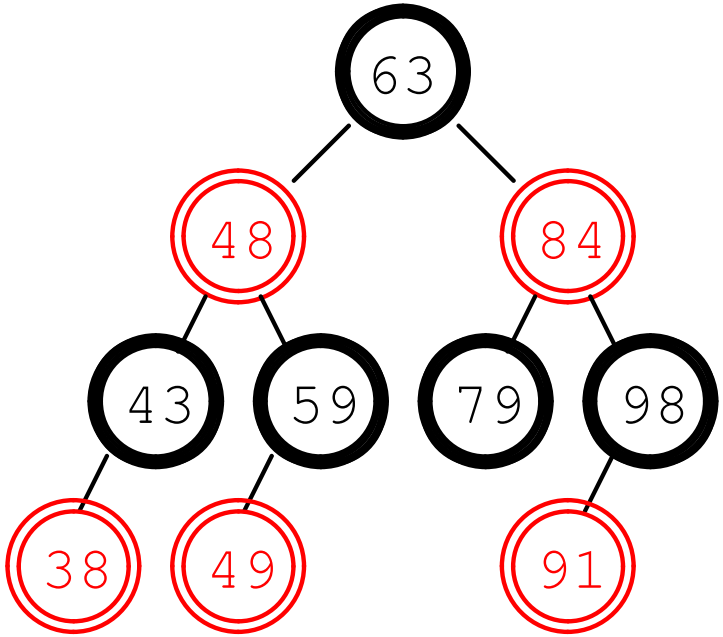
After inserting 98:



the equivalent red-black tree

The red-black tree equivalent to the B-tree is





B-Trees

1. B-Trees
   - nodes with many children
   - a type node
   - a class for B-trees

2. manipulating a B-tree
   - an elaborate example
   - the insertion algorithm
   - removing elements

removing elements

Two cases: item is at leaf or at internal node.

If item is at leaf, then

1. remove the item from the data array,
2. if leaf is less than half full, redistribute,
3. if leaf and its sibling are half full,
   merge leaf, sibling, and parent in one node.

If item is at internal node,
then replace it by its inorder predecessor that is a leaf.

Summary + Exercises

Ended chapter 11 on balancing binary search trees.

Exercises:

1. Generate a random sequence of 16 numbers and insert the numbers into a B-tree where all nodes have 5 children. Draw all intermediate steps.
2. Find the order in which the items were inserted with the first example, that is: what order of 1,2, . . ., 15 gives a tree with all its 4-nodes filled.
3. Write code to count the total number of items stored in a B-tree.