DIVISION OF
DIGITAL CONTENTS
DONGSEO UNIVERSITY

Data Structure

B-Tree

jintaeks@dongseo.ac.kr

December, 2018

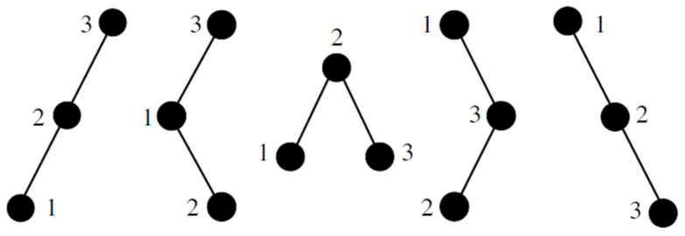
DIVISION OF
DIGITAL CONTENTS
DONGSEO UNIVERSITY

Binary Search Tree

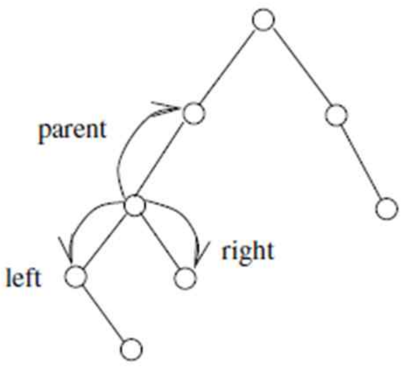
- ✓ **Binary search** requires that we have fast access to *two elements*—specifically the median elements above and below the given node.
- ✓ To combine these ideas, we need a "linked list" with two pointers per node.
  - This is the basic idea behind binary search trees.
- ✓ A ***rooted binary tree*** is recursively defined as either being (1) empty, or (2) consisting of a node called the root, together with two rooted binary trees called **the left and right subtrees**, respectively.

DIVISION OF
DIGITAL CONTENTS
DONGSEO UNIVERSITY

- ✓ A **binary *search* tree** labels each node in a binary tree with a **single key** such that for any node labeled *x*, all nodes in the left subtree of *x* have keys < *x* while all nodes in the right subtree of *x* have keys > *x*.



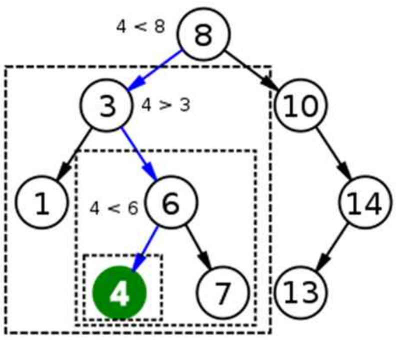
implementing binary search tree



```
typedef struct tree {
    item_type item; // data item
    struct tree* parent; // pointer to parent
    struct tree* left; // pointer to left child
    struct tree* right; // pointer to right child
} tree;
```

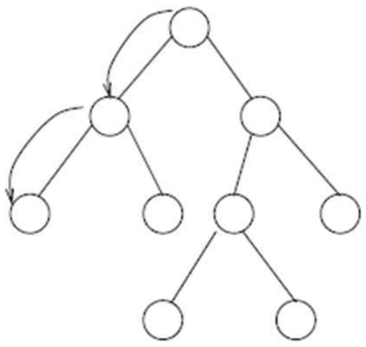
searching in a tree

```
tree *search_tree(tree *l, item_type x)
{
    if (l == NULL) return(NULL);
    if (l->item == x) return(l);
    if (x < l->item)
        return( search_tree(l->left, x) );
    else
        return( search_tree(l->right, x) );
}
```



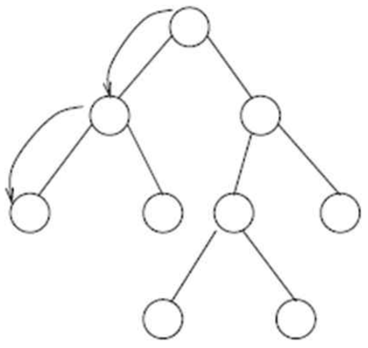
finding minimum element in a tree

```
tree *find_minimum(tree *t)
{
    tree *min; // pointer to minimum
    if (t == NULL) return(NULL);
    min = t;
    while (min->left != NULL)
        min = min->left;
    return(min);
}
```



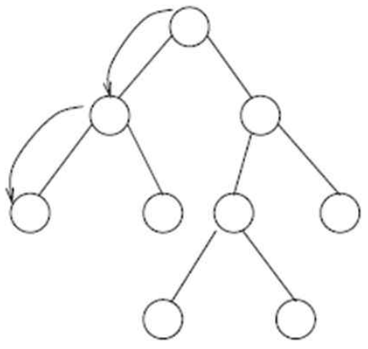
traversing in a tree

```
void traverse_tree(tree *l)
{
    if (l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```



insertion in a tree

```
insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p; /* temporary pointer */
    if (*l == NULL) {
        p = malloc(sizeof(tree)); /* allocate new node
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p; /* link into parent's record */
        return;
    }
    if (x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```

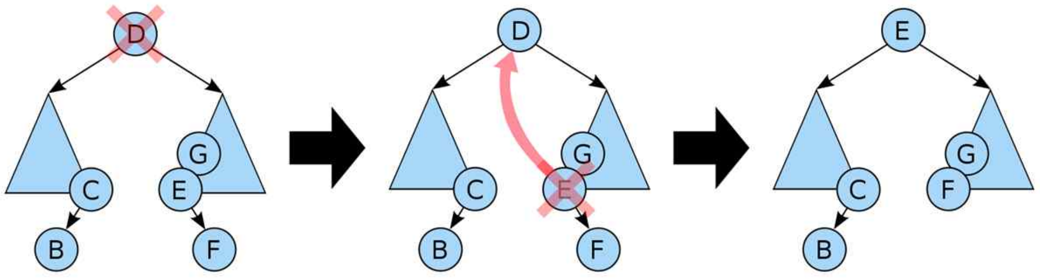


deletion from a tree

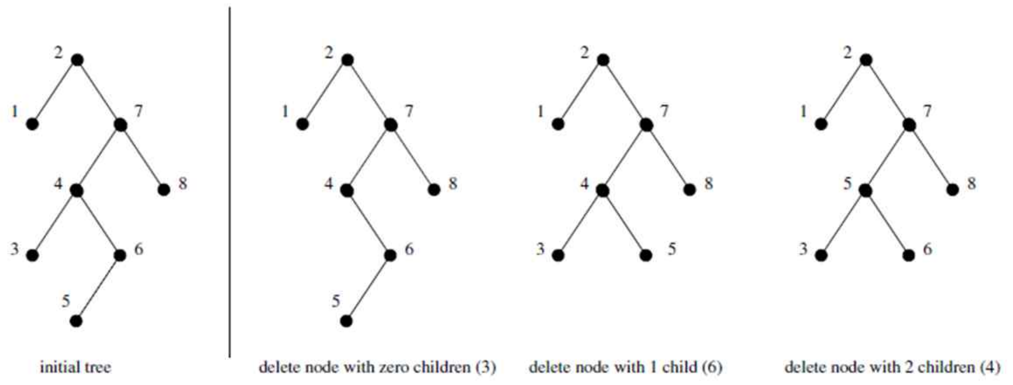
1. Deleting a node with no children: simply remove the node from the tree.
2. Deleting a node with one child: remove the node and replace it with its child.

deletion from a tree

3. Deleting a node with two children:
   - call the node to be deleted *D*.
   - Do not delete *D*. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node *E* (figure).
   - Copy the user values of *E* to *D*.
   - If *E* does not have a child simply remove *E* from its previous parent *G*.
   - If *E* has a child, say *F*, it is a right child. Replace *E* with *F* at *E*'s parent.



ex) deletion



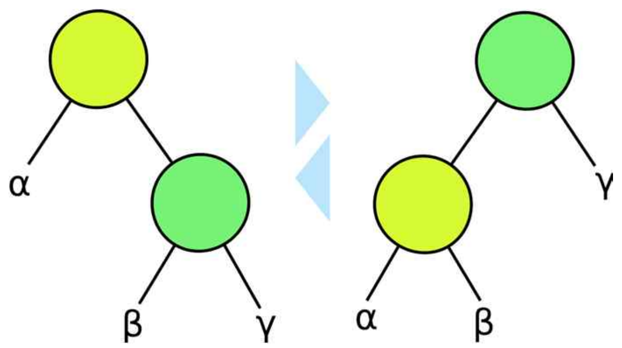
initial tree
delete node with zero children (3)
delete node with 1 child (6)
delete node with 2 children (4)

Figure : Deleting tree nodes with 0, 1, and 2 children

How good are binary search trees?

- ✓ Unfortunately, bad things can happen when building trees through insertion.
- ✓ The data structure has no control over the order of insertion. Consider what happens if the user inserts the keys in sorted order. The operations insert(*a*), followed by insert(*b*), insert(*c*), insert(*d*), . . . will produce **a skinny linear height tree** where only right pointers are used.

Rotation

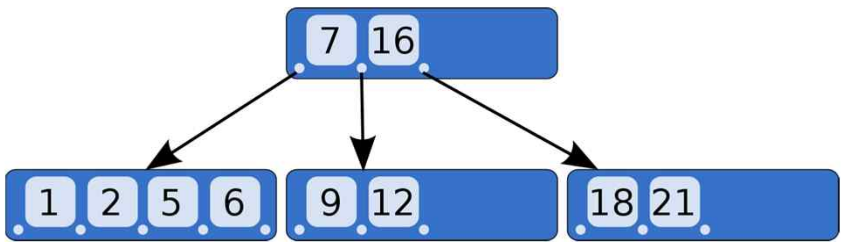


- Tree rotations are very common internal operations in binary trees to keep perfect, or near-to-perfect, internal balance in the tree

B-Tree

✓ A **B-tree** is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.

✓ The B-tree is a generalization of a binary search tree in that a node can have more than two children.

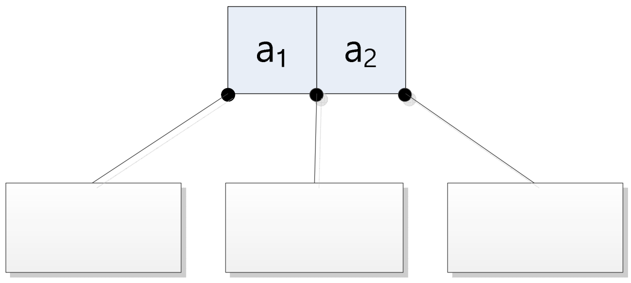


- A B-tree (Bayer & McCreight 1972) of order 5.

- ✓ In B-trees, **internal (non-leaf) nodes** can have a variable number of **child nodes within some pre-defined range**.
- ✓ When data is inserted or removed from a node, its number of child nodes changes.
- ✓ In order to maintain the pre-defined range, internal nodes may be **joined or split**.
- ✓ The lower and upper bounds on **the number of child nodes** are typically **fixed** for a particular implementation.
  - For example, in a 2-3 B-tree(often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

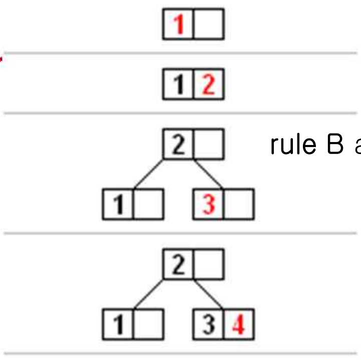
Key and Subtree

- ✓ Each internal node of a B-tree contains a number of keys.
- ✓ The keys act as separation values which divide its subtrees.
- ✓ For example, if **an internal node has 3 child nodes** (or subtrees) then it must have 2 keys: a_1 and a_2.
- ✓ All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2

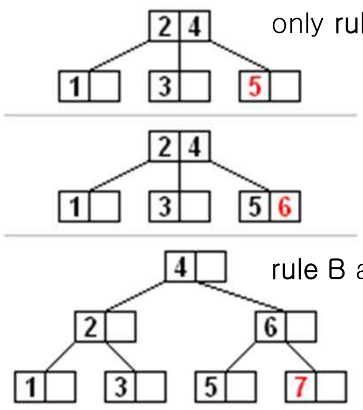


Insertion

- ✓ If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
- ✓ Otherwise the node is full, evenly split it into two nodes so:
  - A **single median** is chosen from among the leaf's elements and the new element.
  - Values less than the median are put in the **new left node** and values greater than the median are put in the **new right node**, with the median acting as a separation value.
  - The separation value is **inserted in the node's parent**, which may cause it to be split, and so on**(rule A)**. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree)**(rule B)**.



rule B applied for '2'



only rule A applied

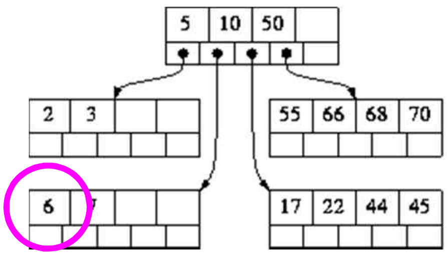
rule B applied for '6'

Deletion

- ✓ To delete value X from a B-tree, starting at a leaf node, there are 2 steps:
  1. Remove X from the current node. Being a leaf node there are no subtrees to worry about.
  2. Removing X might cause the node containing it to have *too few* values.
  - Recall that we require the root to have at least 1 value in it and all other nodes to have at least (M-1)/2 values in them. If the node has too few values, we say it has ***underflowed***.
  - If underflow does not occur, then we are finished the deletion process.
  - If it does occur, it must be fixed. The process for fixing a root is slightly different than the process for fixing the other nodes, and will be discussed afterwards.

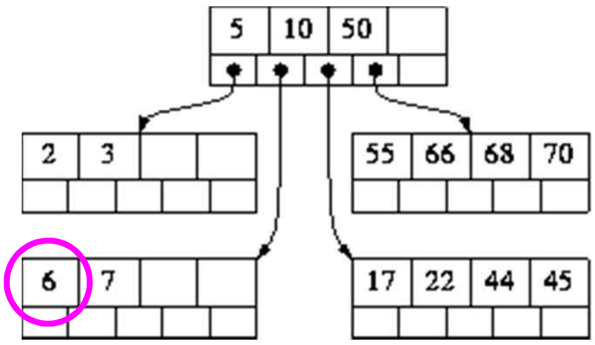
19

- ✓ deleting 6 from this B-tree (of degree 5):



- ✓ Removing 6 causes the node it is in to underflow, as it now contains just one value (7).
- ✓ Our strategy for fixing this is to try to 'borrow' values from a neighbouring node.
- ✓ We join together the current node and its more populous neighbour to form a `combined node' - and we must also include in the combined node the value in the parent node that is in between these two nodes.

20



- ✓ In this example, we join node [7] with its more populous neighbour [17 22 44 45] and put `10' in between them, to create
- ▪ [7 10 17 22 44 45]

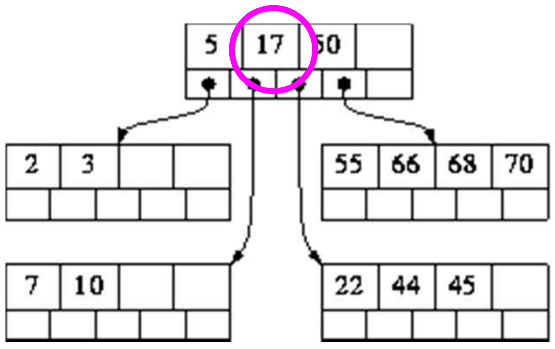
- ✓ How many values might there be in this combined node?
  - The parent node contributes **1** value.
  - The node that underflowed contributes exactly **(M-1)/2-1** values.
  - The neighbouring node contributes somewhere between **(M-1)/2** and (M-1) values.
- ✓ The treatment of the combined node is different depending on whether the neighbouring contributes exactly (M-1)/2 values or more than this number.

- ▪ Left = [7 10]
- ▪ Middle = 17
- ▪ Right = [22 44 45]

Deletion: Case 1

- ✓ Suppose that the neighbouring node contains more than (M-1)/2 values.
- ✓ In this case, the total number of values in the combined node is strictly greater than 1 + ((M-1)/2 - 1) + ((M-1)/2), i.e. it is strictly greater than (M-1).
- ✓ So it must contain M values or more.
- ✓ We split the combined node into three pieces: **Left, Middle, and Right**, where Middle is a single value in the very middle of the combined node.

- ✓ Then put **Middle** into the parent node (in the position where the `10' had been) with **Left** and **Right** as its children.

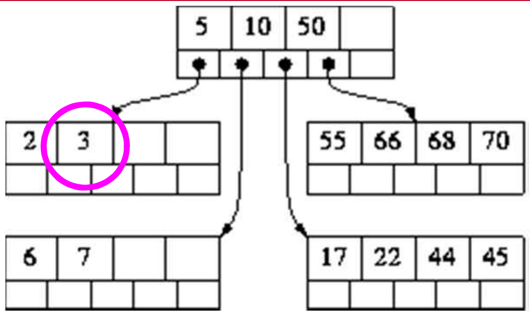


Deletion: Case 2

- ✓ Suppose, on the other hand, that the neighbouring node contains exactly (M-1)/2 values.
- ✓ Then the total number of values in the combined node is 1 + ((M-1)/2 - 1) + ((M-1)/2) = (M-1)
- ✓ In this case the combined node contains the right number of values to **be treated as a node**.
- ✓ So we make it into a node and remove from the parent node the value that has been incorporated into the new, combined node.

25

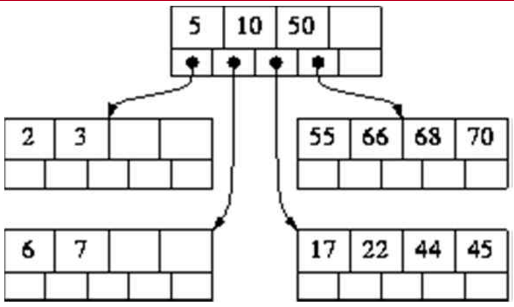
DIVISION OF DIGITAL CONTENTS DONGSEO UNIVERSITY



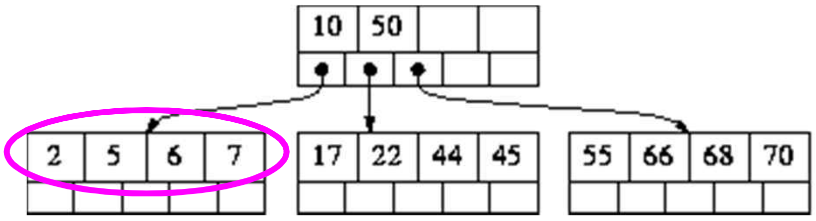
- ✓ We had deleted 3.
- ✓ The node [2 3] underflows when 3 is removed.
- ✓ It would be combined with its more populous neighbour [6 7] and the intervening value from the parent (5) to create the combined node [2 5 6 7].

26

DIVISION OF DIGITAL CONTENTS DONGSEO UNIVERSITY



- ✓ This contains 4 values, so it can be used without further processing.

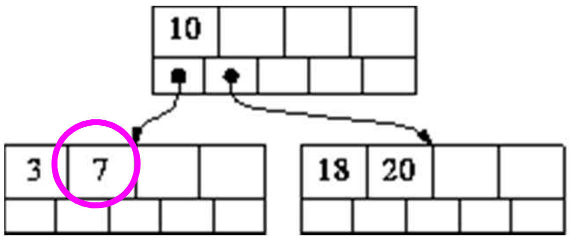


parent node underflows

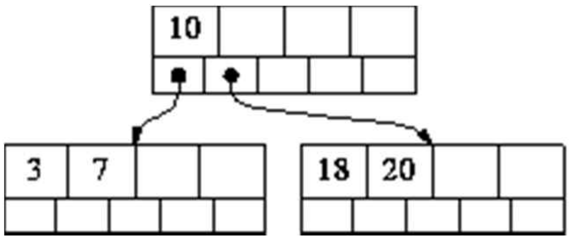
- ✓ It is very important to note that the parent node now has one fewer value.
- ✓ This might cause *it* to underflow - imagine that 5 had been the *only* value in the parent node.
- ✓ **If the parent node underflows**, it would be treated in exactly the same way - combined with *it*s more populous neighbour etc.
- ✓ The underflow processing repeats at successive levels until no underflow occurs **or until the root underflows** (the processing for root-underflow is described next).

root node

- ✓ If the root's two children are now a single node, then *that node* can be used as the new root, and the current root (which has underflowed) can simply be deleted.
- ✓ We delete 7 from this B-tree (M=5):

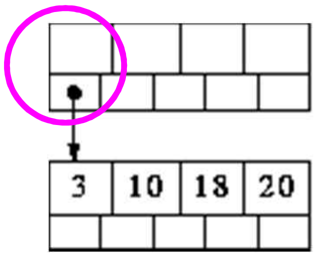


- ✓ We delete 7 from this B-tree (M=5):



- ✓ The node [3 7] would underflow, and the combined node [3 10 18 20] would be created.
- ✓ This has 4 values, which is acceptable when M=5

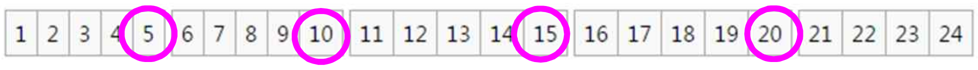
- ✓ So it would be kept as a node, and `10' would be removed from the parent node - the root.
- ✓ This is the only circumstance in which underflow can occur in a root that is not a leaf. The situation is this:



- ✓ the current root node, now empty, can be deleted and its child used as the new root

Initial construction

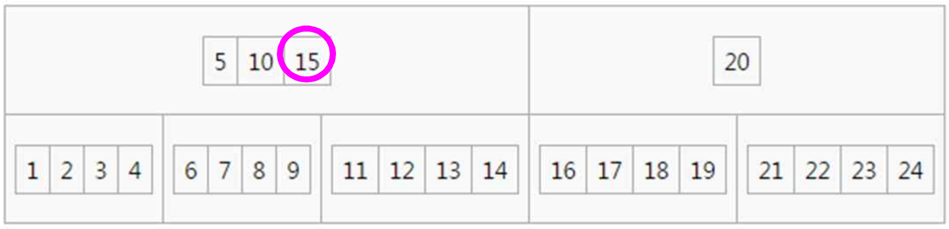
- ✓ For example, if **the leaf nodes have maximum size 4** and the initial collection is the integers 1 through 24, we would initially construct 4 leaf nodes containing 5 values each and 1 which contains 4 values:



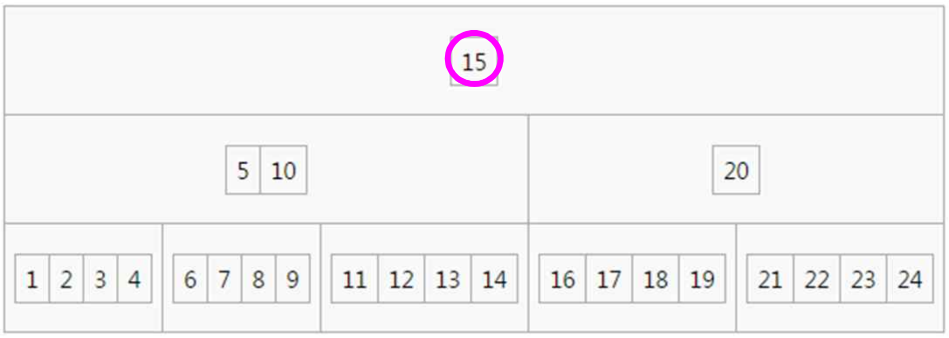
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

- ✓ suppose the internal nodes contain at most 2 values (**3 child pointers**).

- ✓ We build the next level up from the leaves by taking the last element from each leaf node except the last one.
- ✓ Again, each node except the last will contain one extra value. In the example, suppose the internal nodes contain at most 2 values (3 child pointers).

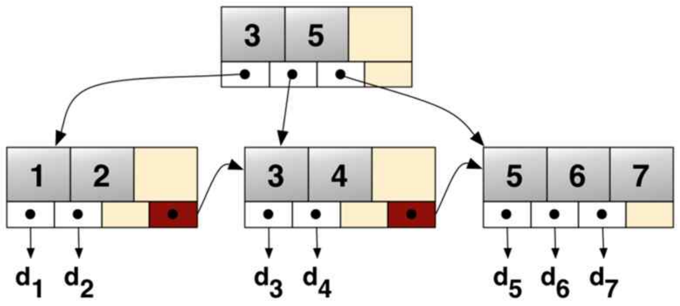


- ✓ This process is continued until we reach a level with only one node and it is not overfilled.



B+ Tree

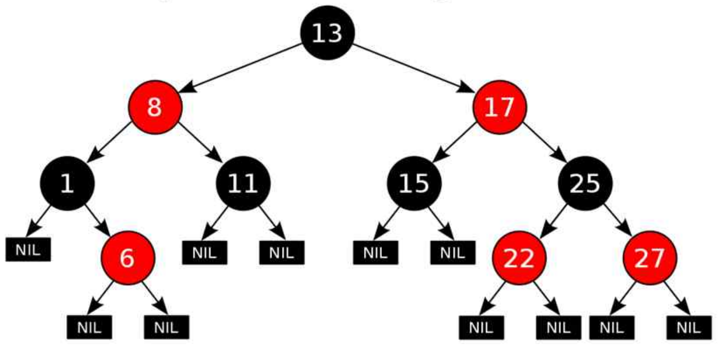
- ✓ In the B+ tree, copies of the keys are stored in the internal nodes; **the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access**.



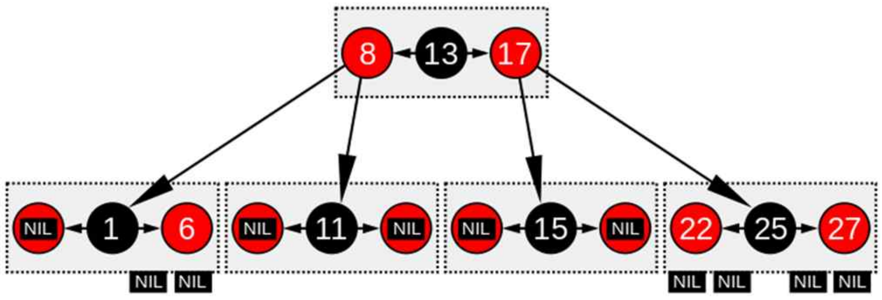
- A simple B+ tree example linking the keys 1–7 to data values d_1-d_7. The linked list (red) allows rapid in-order traversal

Red-black tree

- ✓ A **red–black tree** is a kind of self-balancing binary search tree in computer science.
- ✓ Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
- ✓ These color bits are used to ensure **the tree remains approximately balanced** during insertions and deletions



- ✓ Tracking the color of each node requires only 1 bit of information per node because there are only two colors.
- ✓ The tree does not contain any other data specific to its being a red–black tree so its **memory footprint is almost identical to a classic (uncolored) binary search tree**.
  - – In many cases, the additional bit of information can be stored at no additional memory cost.



- ✓ A red–black tree is similar in structure to a B-tree of order 4, where each node can contain between 1 and 3 values and (accordingly) between 2 and 4 child pointers

Ex) set::set<int>

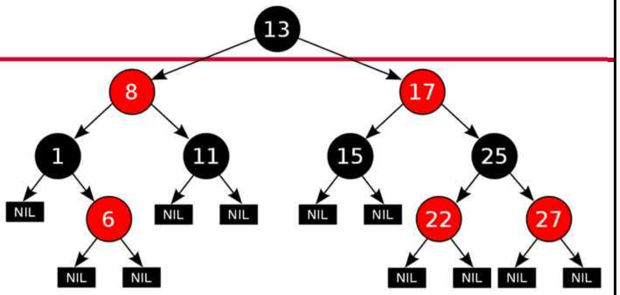
```
#include <stdio.h>
#include <set>

void main()
{
    std::set<int> s{ 1, 3, 5, 7, 9 };
    for (std::set<int>::iterator it = s.begin(); it != s.end(); ++it)
    {
        const int value = *it;
        printf("%i, ", value);
    }
}
```

39

DIVISION OF DIGITAL CONTENTS DONGSEO UNIVERSITY

std::set<T>



```
_Myiter& operator++()
{// preincrement
    if (_Mytree::_Isnil(_Ptr))
        ;// end() shouldn't be incremented, don't move
    else if (!_Mytree::_Isnil(_Mytree::_Right(_Ptr)))
        _Ptr = _Mytree::_Min(
            _Mytree::_Right(_Ptr));// ==> smallest of right subtree
    else {// climb looking for right subtree
        _Nodeptr _Pnode;
        while (!_Mytree::_Isnil(_Pnode = _Mytree::_Parent(_Ptr))
            && _Ptr == _Mytree::_Right(_Pnode))
            _Ptr = _Pnode;// ==> parent while right subtree
        _Ptr = _Pnode;// ==> parent (head if end())
    }
    return (*this);
}
```

40

DIVISION OF DIGITAL CONTENTS DONGSEO UNIVERSITY

ex) std::map

```
#include <iostream>
#include <map>

int main()
{
    std::map<int,char> example = {{1,'a'},{2,'b'}};

    auto search = example.find(2);
    if(search != example.end()) {
        std::cout << "Found " << search->first << " " << search->second << '₩n';
    }
    else {
        std::cout << "Not found₩n";
    }
}
```

References

- ✓ https://en.wikipedia.org/wiki/B-tree
- ✓ Skiena, The Algorithm Design Manual
- ✓ https://webdocs.cs.ualberta.ca/~holte/T26/del-b-tree.html

MY BRIGHT FUTURE

DSU Dongseo University 동서대학교