

## Lecture 04

- Semi Event-Driven programming
- Binary Tree
  - Constructing a binary tree
  - Visualizing a binary tree
  - Deleting a binary-tree node
  - Binary-tree rotation
  - Re-balancing binary tree
- Problem Set 1: Rebalancing a Binary Tree

## Semi Event-Driven Programming

### Polling-based programming vs. Event-Driven programming

- Polling-based programming
  - The application actively checks for the events such as key strokes, mouse movement, finger touch, etc.
  - The application has its own main loop.
- Event-driven programming
  - The application can do something only when it is called from the operating system upon events.
  - To move something continuously, the application uses an timer or interval event.
  - The application must not hold on to its main loop. The event-handling function must return as soon as one step of the process is done.

- Polling-based programming is in general easier because you control in what order the events are checked and processed.
- In event-driven programming you cannot expect in what order the events are thrown at your program.

## Semi Event-Driven programming framework

- FsLazyWindow framework.
- Assuming your directory structure is:

~hummingbird

eng\_comp

src

public

hummingbird

(\* Replace hummingbird with your Andrew ID)

1. Cd or pushd to ~/eng\_comp/src/hummingbird (%USERPROFILE%\eng\_comp\src\hummingbird in Windows)
2. Type:  

```
svn export ../public/src/fslazywindow/template lazy_window_app
```
3. Edit CMakeLists.txt in lazy\_window\_app, and change TARGET\_NAME as:  

```
set(TARGET_NAME lazy_window_app)
```
4. Edit top-level CMakeLists.txt and add the following line at the bottom:  

```
add_subdirectory(hummingbird/lazy_window_app)
```
5. Re-run CMake and build lazy\_window\_app

In this framework, you need to implement...

- `FsLazyWindowApplication` class
- `FsLazyWindowApplication::FsLazyWindowApplication()`
- `void FsLazyWindowApplication::BeforeEverything(int argc, char *argv[])`
- `void FsLazyWindowApplication::GetOpenWindowOption(FsOpenWindowOption &opt) const`
- `void FsLazyWindowApplication::Initialize(int argc, char *argv[])`
- `void FsLazyWindowApplication::Interval(void)`
- `void FsLazyWindowApplication::BeforeTerminate(void)`
- `void FsLazyWindowApplication::Draw(void)`
- `bool FsLazyWindowApplication::UserWantToCloseProgram(void)`
- `bool FsLazyWindowApplication::MustTerminate(void) const`
- `long long int FsLazyWindowApplication::GetMinimumSleepPerInterval(void) const`
- `bool FsLazyWindowApplication::NeedRedraw(void) const`

Many of them can be left empty, or can use the template as is. But, you need to implement `Interval` and `Draw` to get something meaningful.

## The sequence

- When the program starts, the framework calls BeforeEverything with command-parameters.
- Then the framework will ask your program about the location, size, and options of the window by GetOpenWindowOption. The information you return may be ignored (for example in iOS)
- Initialize function is called after the window and OpenGL context is ready.
- While the program is running, the framework calls Interval and Draw regularly. The program must be prepared to react to these function calls any time.



- When the program needs to close, return true from MustTerminate function. The framework will end the main loop and close.
- The framework calls Draw function when NeedRedraw function returns true. But, Draw function may be called for other reasons, such as when window is re-sized.
- In Interval, you can use functions from FsSimpleWindow framework (FsInkey, FsGetMouseEvent, etc.) to get device state.

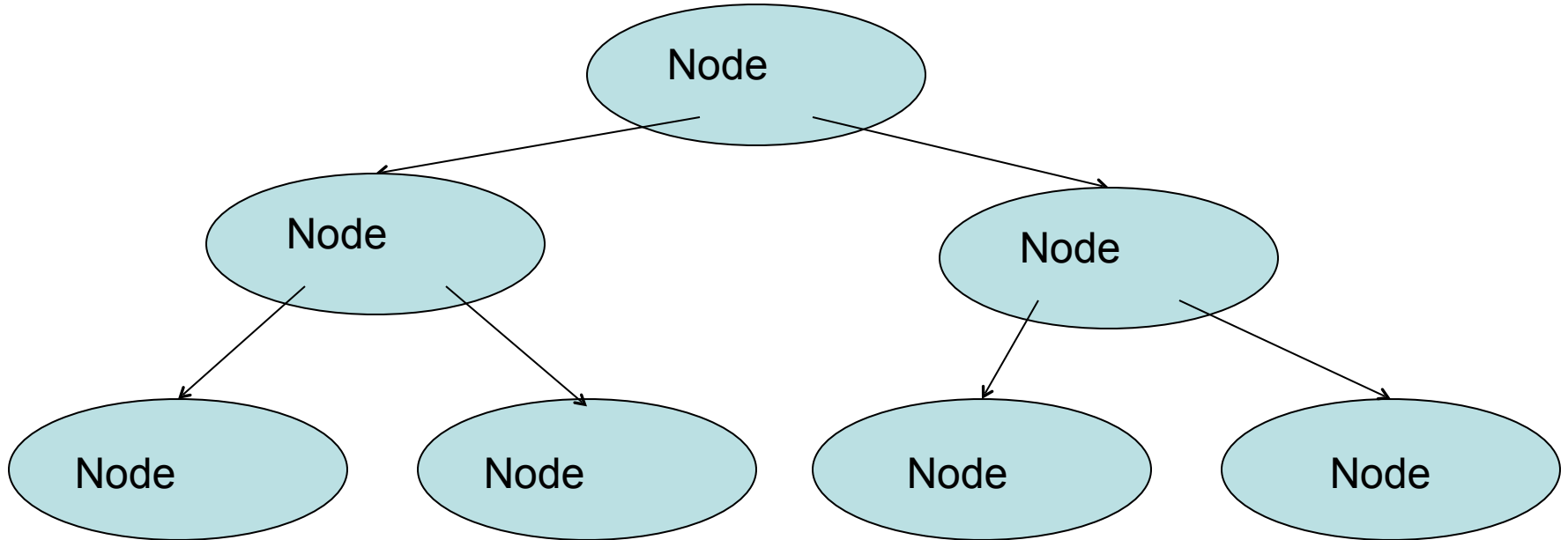
- You can use the following two functions to get touch state:
  - `int FsGetNumCurrentTouch(void);`
  - `const FsVec2i *FsGetCurrentTouch(void);`
- Touch is currently available only on iOS devices and Windows with multi-touch screen.
- Standard for touch interface is volatile. New 3D touch of iOS devices adds a new interface design, but is not clear how the API should be prepared for.
- Building for iOS will be covered later in the semester.

## Transition from Polling-based program to Event-driven program

- All information that can describe the program state must be in the `FsLazyWindowApplication` class.
- Let's do it with the bouncing-ball example.

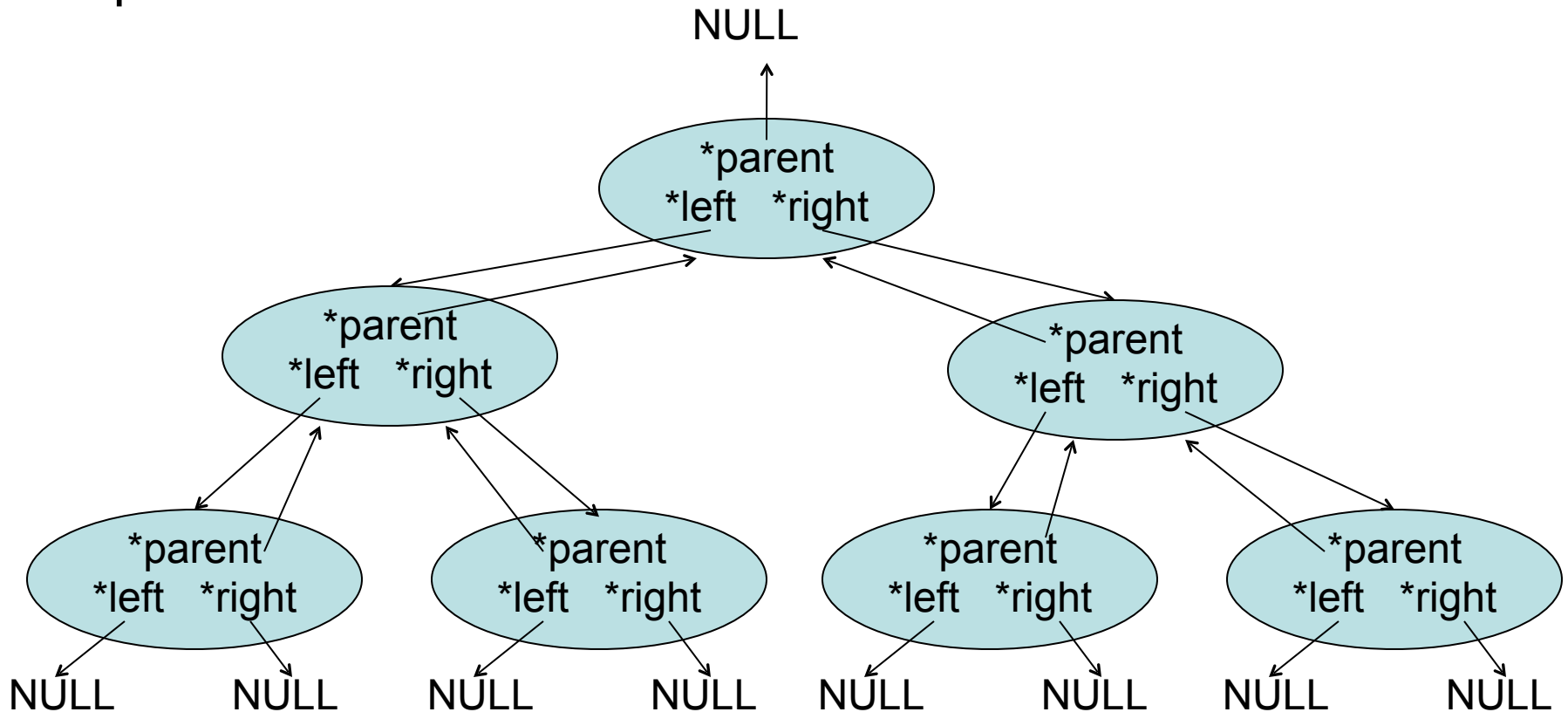
# Binary tree

- Main purposes
  - Efficiently sort objects
  - Efficiently find an object.
  - Very useful when you need a priority queue.



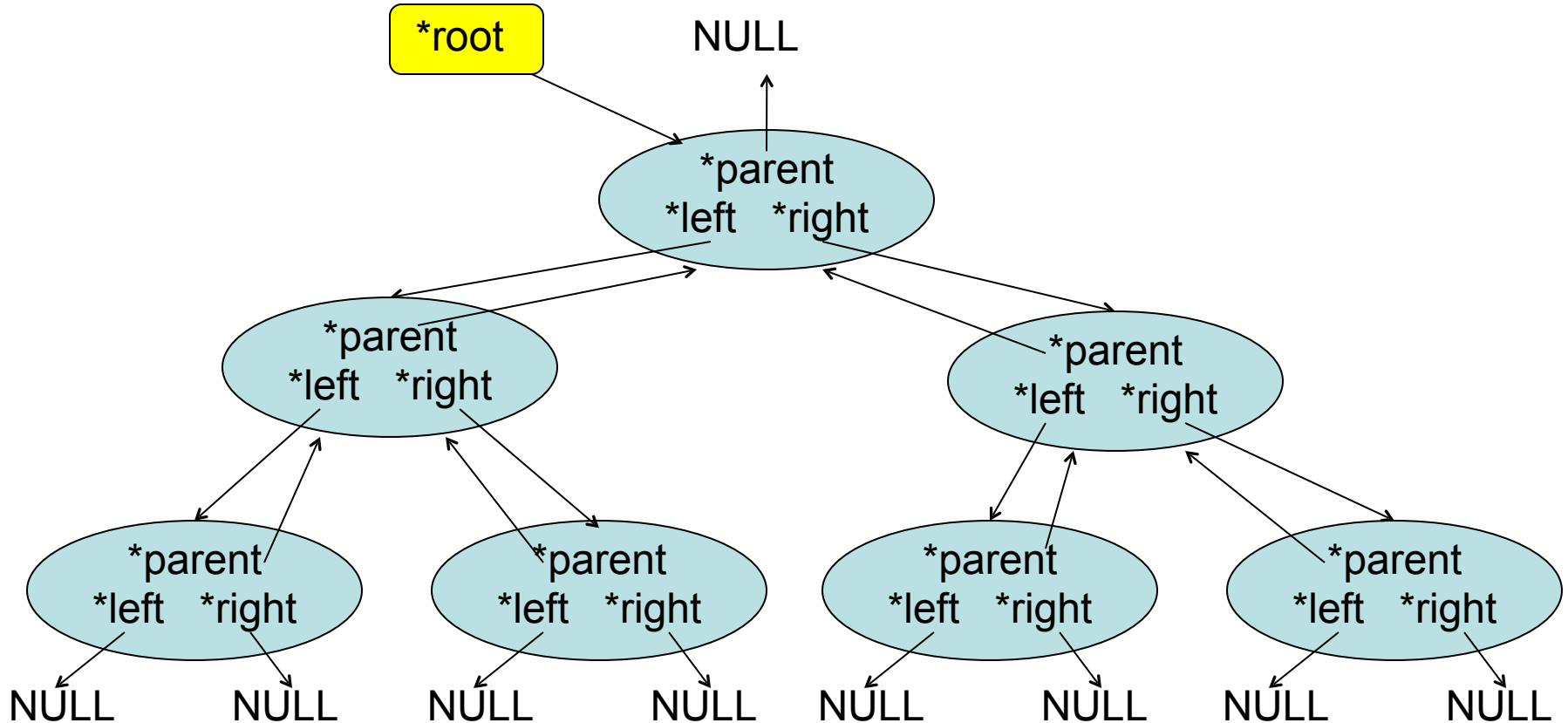
## Binary tree

- Linked list makes a linear connection between objects.
- Binary tree makes a tree-like structure.
- Each node has pointers - left, right, and can also have parent.



## Binary tree

- Just like a linked list is retained by head and tail pointers, a binary tree is retained by a pointer for the root node.

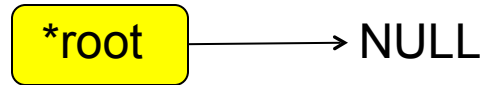


## Binary tree

- Most common usage of a binary tree is for sorting and finding objects quickly.
- Each tree node (someone calls it a leaf) needs to be comparable.

## Binary tree

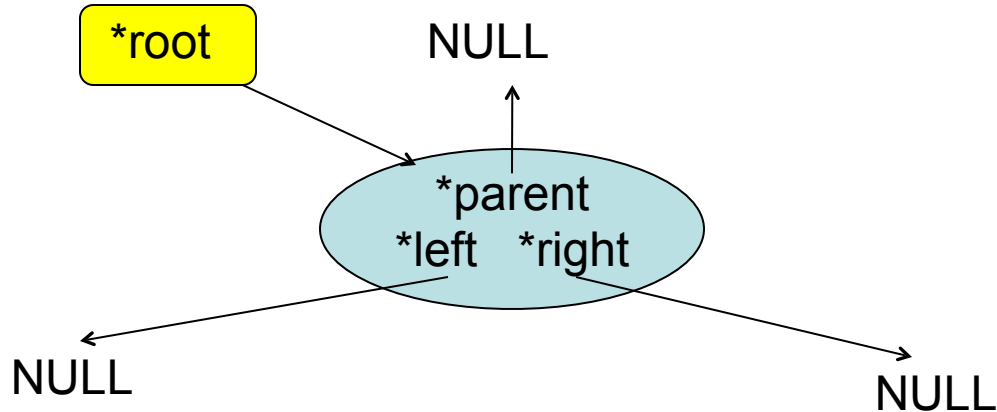
- Initially, the root pointer is NULL. (When the tree is empty.)





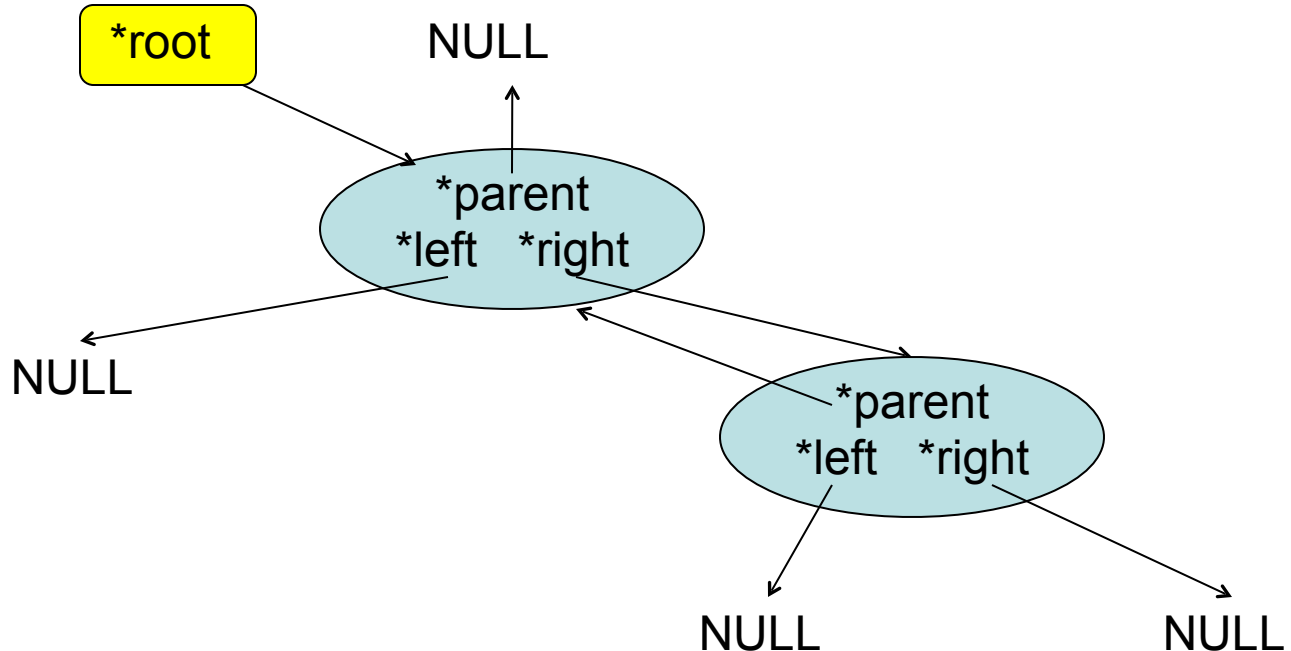
## Binary tree

- When the first node is added, root pointer points to the first node, and the first node's parent, left, and right are all NULL.



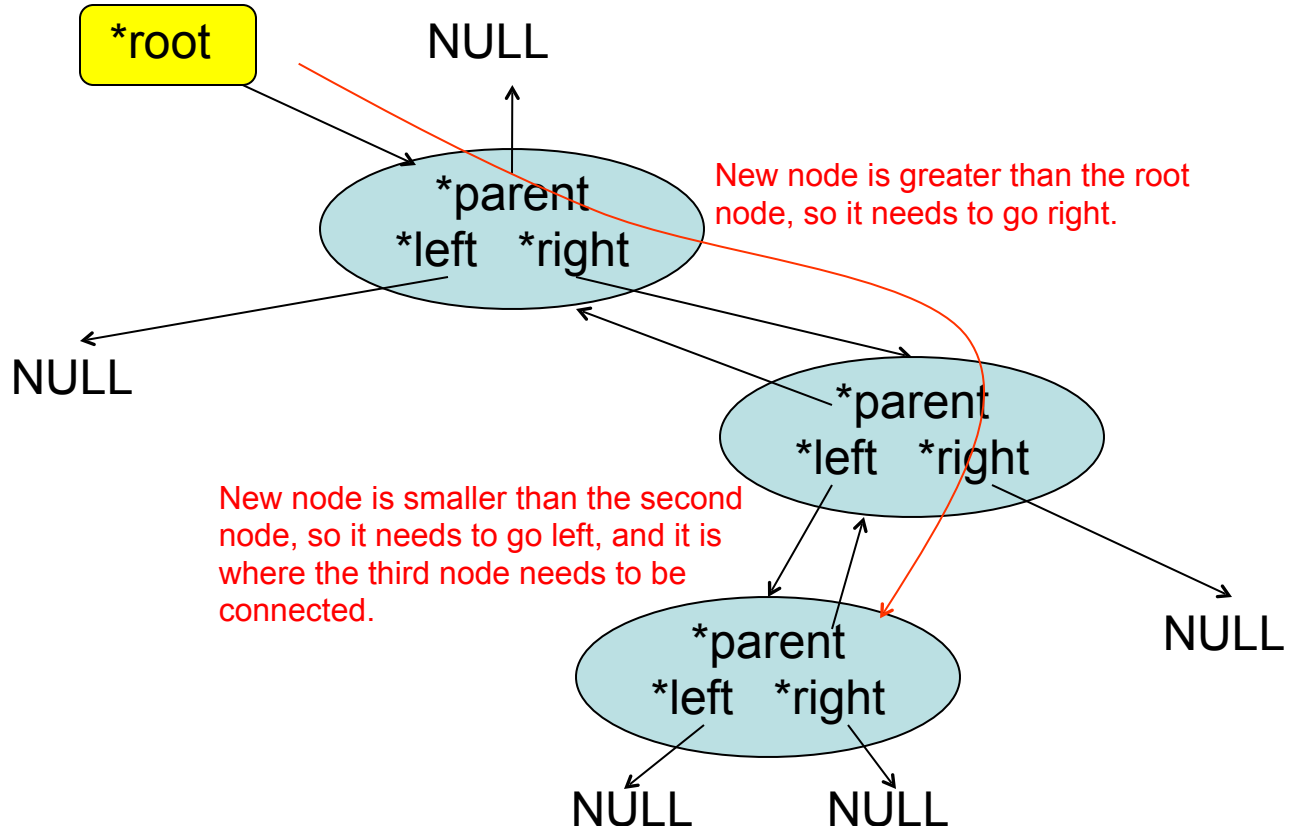
## Binary tree

- When the second node is added (let's say the second is greater than the first node) it is added to the right of the first node.



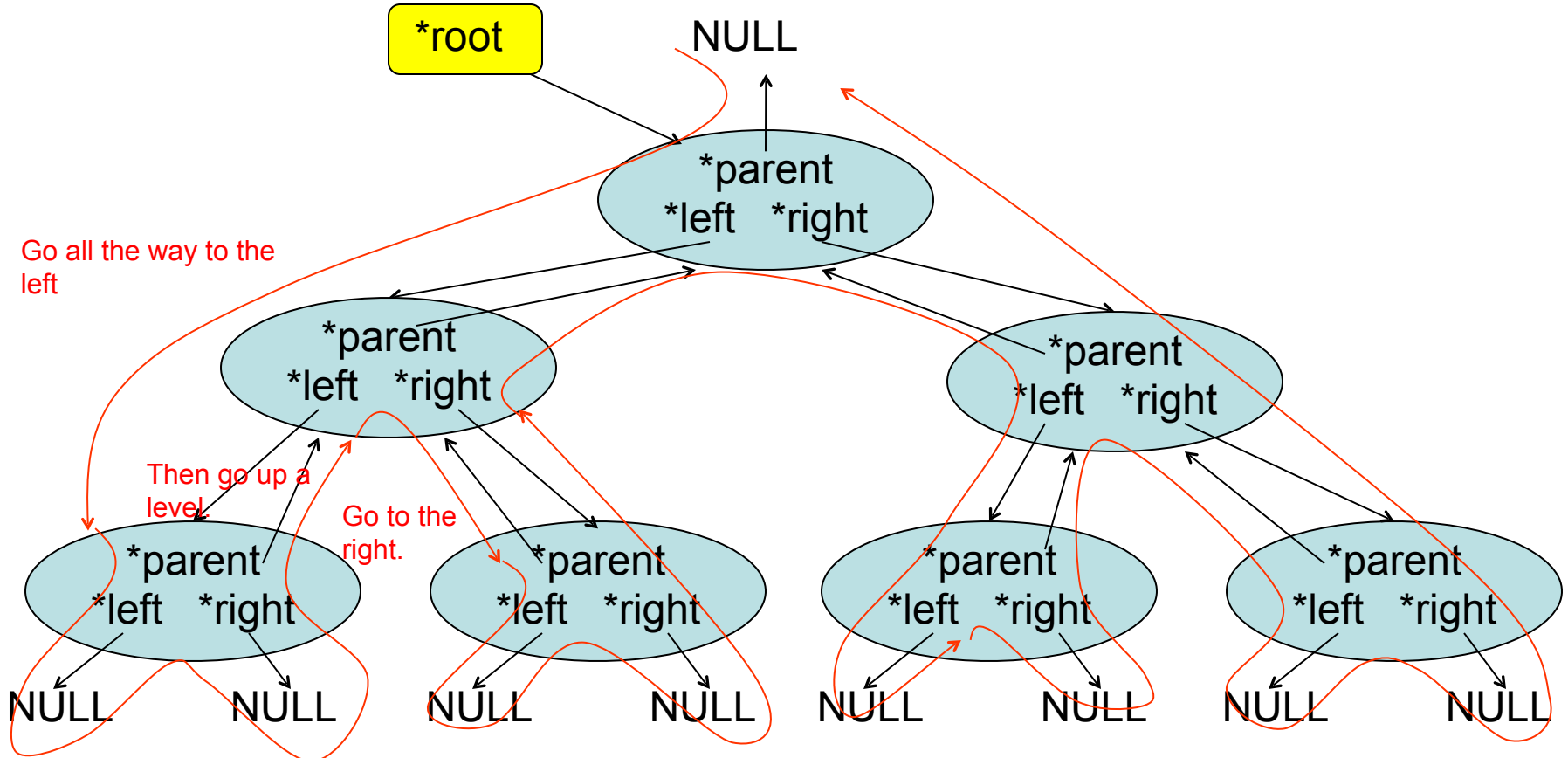
## Binary tree

- Let's say the third node is greater than the first, but smaller than the second node.
- It needs to be added to the left of the second node.



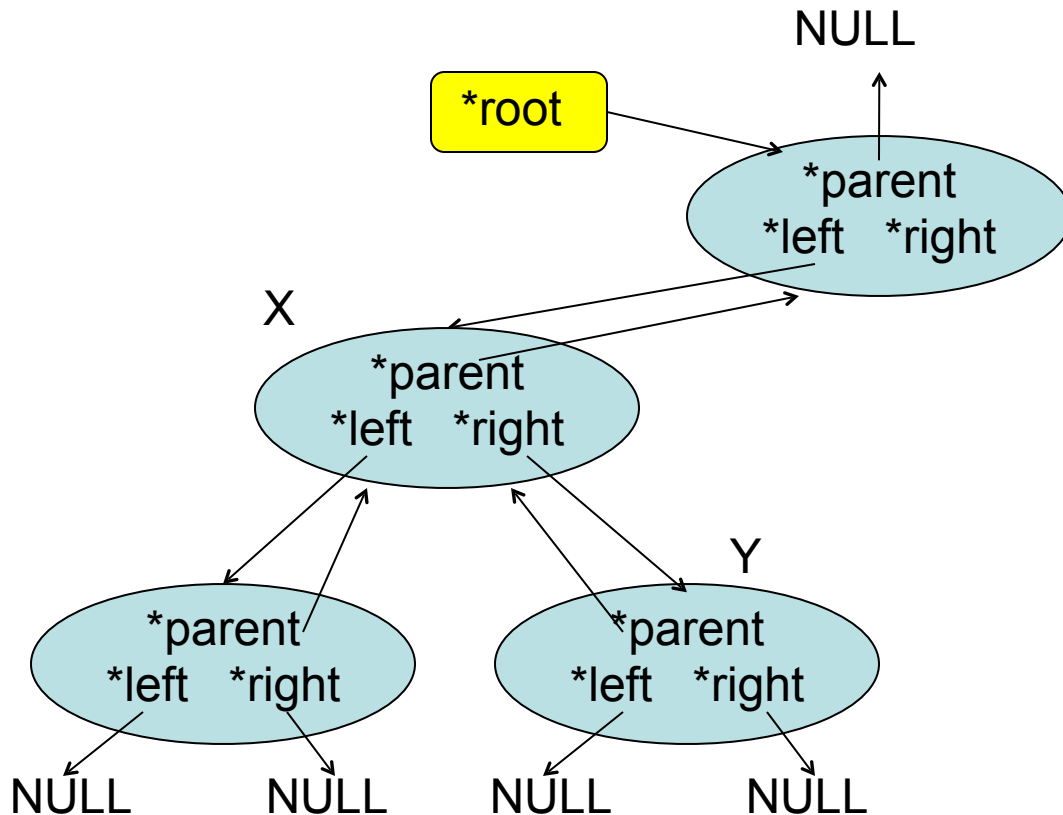
# Binary tree

- After building a whole tree, you can iterate through the smallest node to the greatest node as follows.



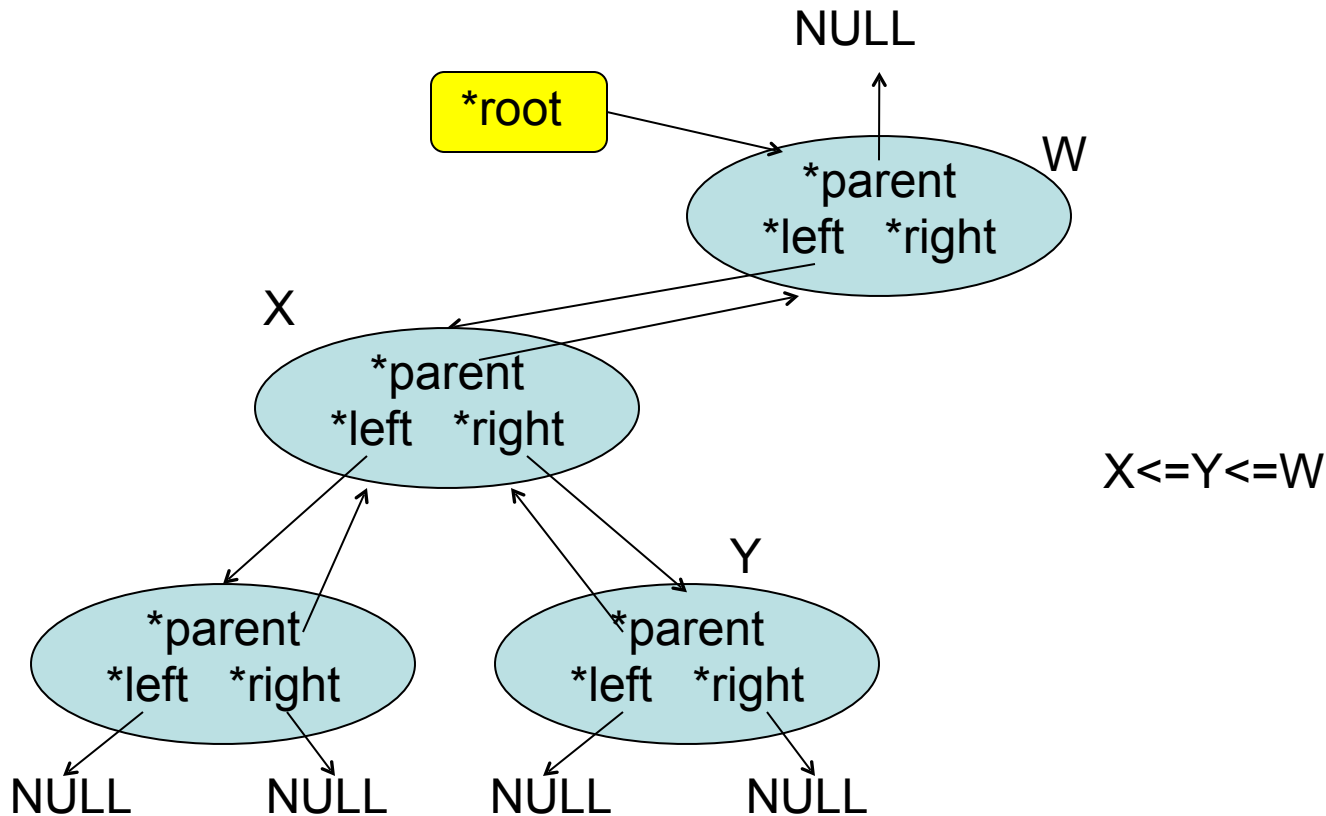
## Binary tree

- Any lower nodes connected to the right (left) of the node X has greater (smaller) nodal values than X.
- In this case, node Y or any lower node of Y has a greater nodal value than X.



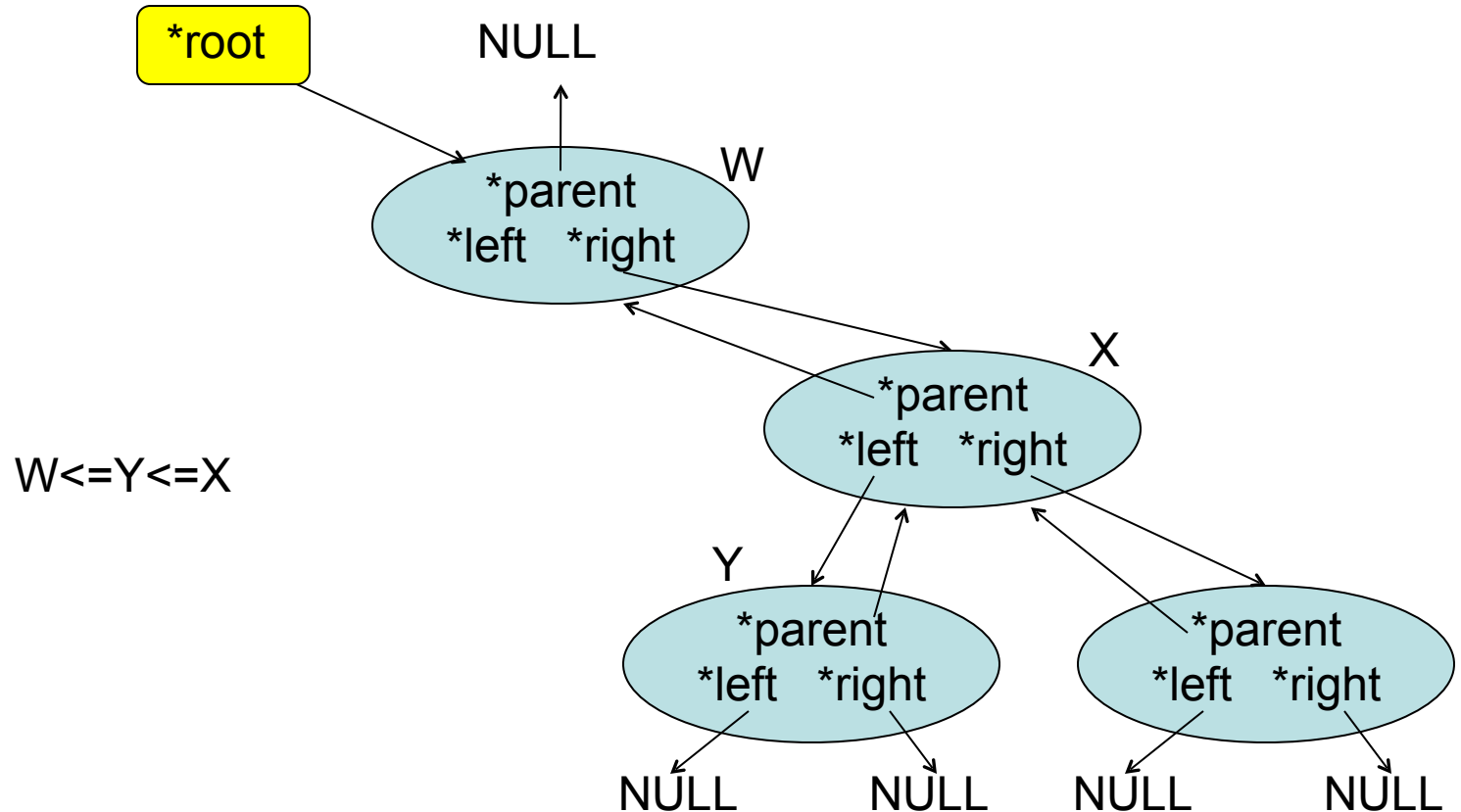
## Binary tree

- If X is immediate left of W, any node connected to the right of X has a nodal value between X and W.



## Binary tree

- Or, if X is immediate right of W, any lower node connected to the left of X has a nodal value between W and X.



## Binary tree

- Finding the first node:
  - Starting from the root node, descend to the left until no more left node is found.
- Finding the next node from the current node:
  - If the node has a right node, move to the right, and then descend all the way to the left until no more left node is found.
  - Otherwise,
    - (1) Move to the parent node.
    - (2) If the current node becomes nullptr, it is the end of the tree.
    - (3) If the previous node is the left of the current node, done.
    - (4) If the previous node is the right of the current node, repeat (1)

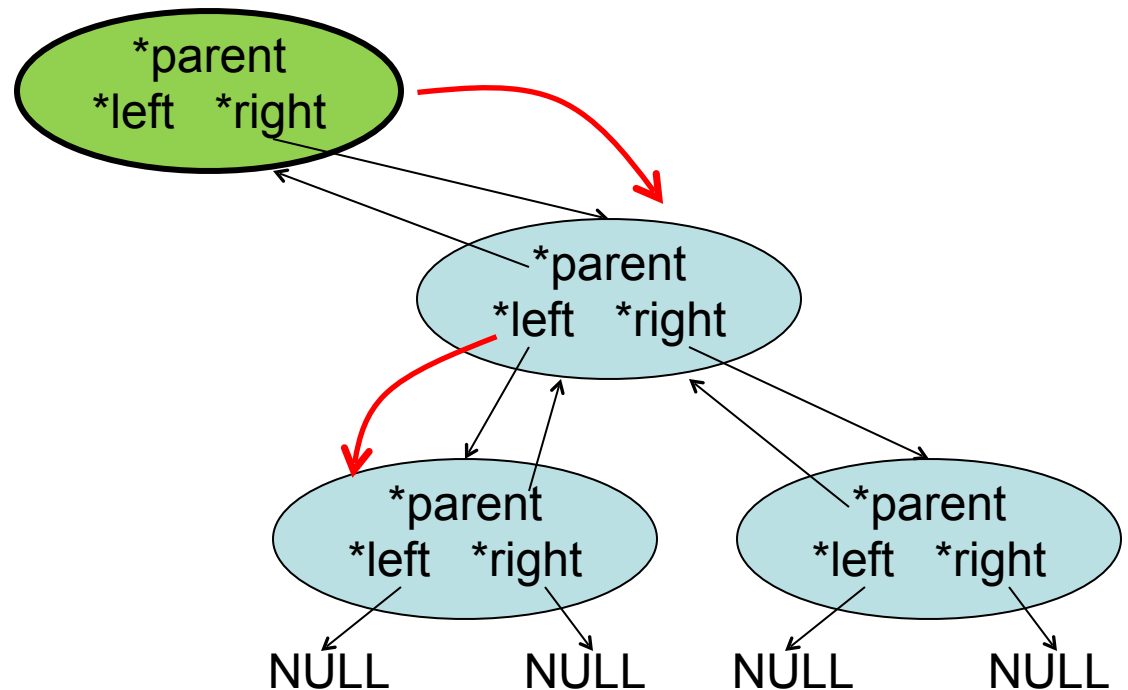


## Finding the next node

If the current node has a non-null right node,

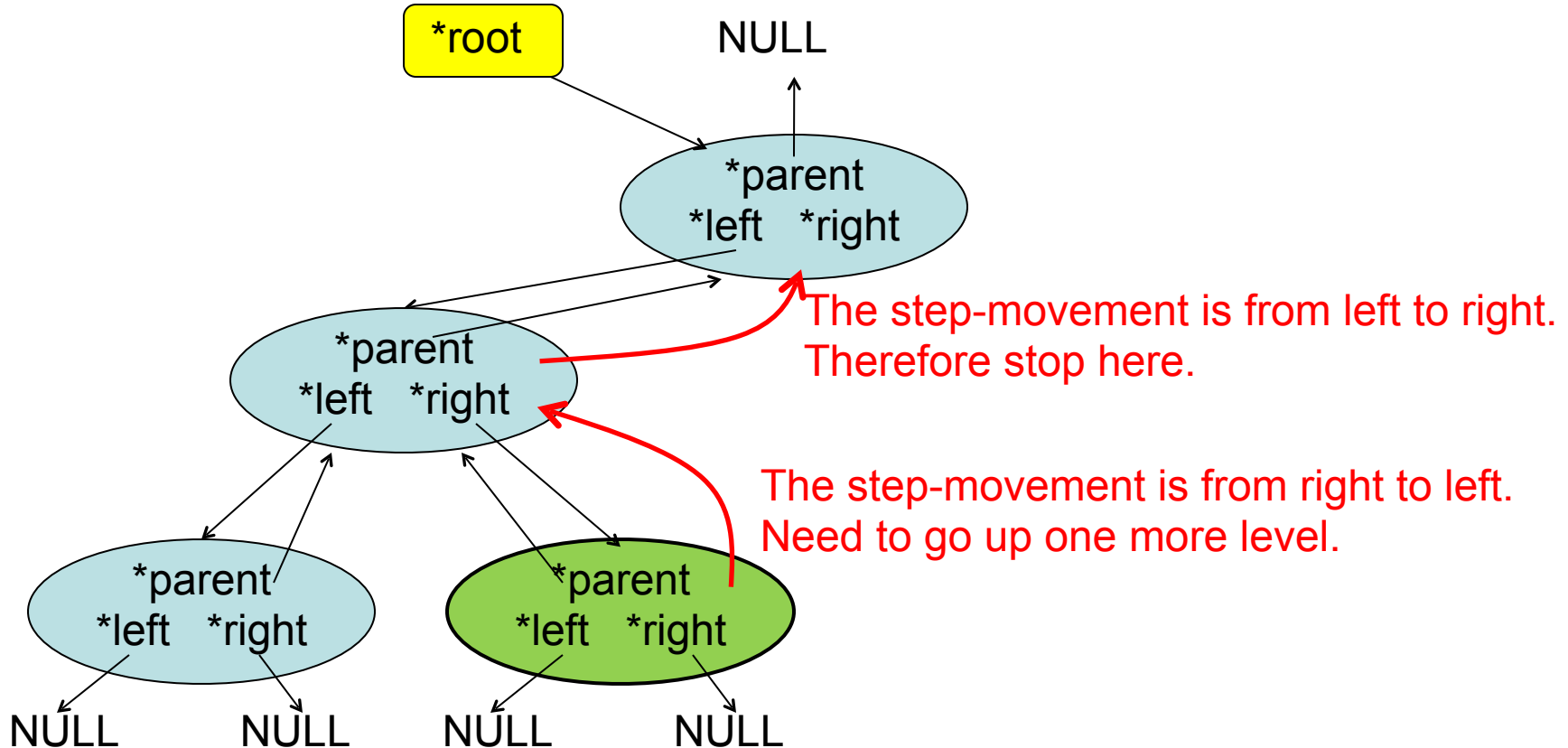
(1) Move right, and then

(2) Move left until no more left node exists.



## Finding the next node

- If the current node does not have a non-null right node,  
(1) Go up until the current node becomes nullptr, or the step-movement was from the left to right.



# Binary tree of an integer

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class BinTreeNode
{
public:
    int key;
    BinTreeNode *parent,*left,*right;
    BinTreeNode()
    {
        parent=nullptr;
        left=nullptr;
        right=nullptr;
    }
};

void Traverse(BinTreeNode *node)
{
    if(nullptr!=node)
    {
        Traverse(node->left);
        printf(" %d",node->key);
        Traverse(node->right);
    }
}
```

```
BinTreeNode *AddBinTreeNode(BinTreeNode *newNode,BinTreeNode *rootNode)
{
    if(nullptr==rootNode)
    {
        rootNode=newNode;
        return rootNode;
    }
    else
    {
        BinTreeNode *seeker=rootNode;
        for(;;)
        {
            if(newNode->key<seeker->key)
            {
                if(nullptr!=seeker->left)
                {
                    seeker=seeker->left;
                }
                else
                {
                    seeker->left=newNode;
                    newNode->parent=seeker;
                    return rootNode;
                }
            }
            else
            {
                if(nullptr!=seeker->right)
                {
                    seeker=seeker->right;
                }
                else
                {
                    seeker->right=newNode;
                    newNode->parent=seeker;
                    return rootNode;
                }
            }
        }
    }
    return nullptr; // Not supposed to come here.
}
```

# Templated version

```
template <class KeyClass,class ValueClass>
class BinTreeNode
{
public:
    KeyClass key;
    ValueClass value;
    BinTreeNode *parent,*left,*right;
    BinTreeNode()
    {
        parent=nullptr;
        left=nullptr;
        right=nullptr;
    }
};
```

Binary-tree is often used for finding a key-value pair. For this purpose, each node must carry a key and a value.

```
template <class KeyClass,class ValueClass>
BinTreeNode <KeyClass,ValueClass> *AddBinTreeNode(
    BinTreeNode <KeyClass,ValueClass> *newNode,
    BinTreeNode <KeyClass,ValueClass> *rootNode)
{
    if(nullptr==rootNode)
    {
        rootNode=newNode;
        return rootNode;
    }
    else
    {
        auto *seeker=rootNode;
        for(;;)
        {
            if(newNode->key<seeker->key)
            {
                if(nullptr==seeker->left)
                {
                    seeker->left=newNode;
                    break;
                }
                seeker=seeker->left;
            }
            else
            {
                if(nullptr==seeker->right)
                {
                    seeker->right=newNode;
                    break;
                }
                seeker=seeker->right;
            }
        }
        newNode->parent=seeker;
        return rootNode;
    }
    return nullptr; // Not supposed to come here.
}
```

# Better protected version

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

template <class KeyClass,class ValueClass>
class BinaryTree
{
public:
    class Node
    {
    public:
        KeyClass key;
        ValueClass value;
        Node *parent,*left,*right;
        Node()
        {
            parent=nullptr;
            left=nullptr;
            right=nullptr;
        }
    };

protected:
    Node *rootNode;

public:
    BinaryTree()
    {
        rootNode=nullptr;
    }
    ~BinaryTree()
    {
        CleanUp();
    }
    const Node *GetRoot(void) const
    {
        return rootNode;
    }
};
```

```
private:
    void Free(Node *ptr)
    {
        if(nullptr!=ptr)
        {
            Free(ptr->left);
            Free(ptr->right);
            delete ptr;
        }
    }

public:
    void CleanUp(void)
    {
        Free(rootNode);
        rootNode=nullptr;
    }

private:
    void Add(Node *newNode);
public:
    const Node *Add(KeyClass key,const ValueClass &value);
};
```

## Better protected version (Continued)

```
//
template <class KeyClass,class ValueClass>
void BinaryTree <KeyClass,ValueClass>::Add(
    typename BinaryTree <KeyClass,ValueClass>::Node *newNode)
{
    if(nullptr==rootNode)
    {
        rootNode=newNode;
    }
    else
    {
        auto *seeker=rootNode;
        for(;;)
        {
            if(newNode->key<seeker->key)
            {
                if(nullptr==seeker->left)
                {
                    seeker->left=newNode;
                    break;
                }
                seeker=seeker->left;
            }
            else
            {
                if(nullptr==seeker->right)
                {
                    seeker->right=newNode;
                    break;
                }
                seeker=seeker->right;
            }
        }
        newNode->parent=seeker;
    }
}
```

```
template <class KeyClass,class ValueClass>
const typename BinaryTree<KeyClass,ValueClass>::Node *
    BinaryTree<KeyClass,ValueClass>::Add(
        KeyClass key,const ValueClass &value)
{
    auto newNode=new BinaryTree<KeyClass,ValueClass>::Node;
    newNode->key=key;
    newNode->value=value;
    Add(newNode);
    return newNode;
}

// Test functions >>
void Traverse(const BinaryTree <int,int>::Node *node)
{
    if(nullptr!=node)
    {
        Traverse(node->left);
        printf(" %d",node->key);
        Traverse(node->right);
    }
}

int main(void)
{
    srand((int)time(nullptr));
    BinaryTree <int,int> bTree;
    for(int i=0; i<10; ++i)
    {
        bTree.Add(rand()%100,0);
    }
    Traverse(bTree.GetRoot());
    return 0;
}

// Test functions <<
```

Typename is a bit confusing. Whenever the compiler identifies that the type can cause an ambiguity, the compiler requires to add a keyword "typename" in front of the data type. Very conservative C++ specification (because compiler can tell typename is required or not.) When you get an error "typename required"

## Visualizing a binary tree

### Goal:

- Visualize how the binary-tree is constructed.
- Interactively apply a binary-tree operation called tree rotation.
- Implement and verify tree-rebalancing algorithm.

## Visualizing a Binary-Tree

- Create a copy of FsLazyWindow template and name it binTreeVis.  
`svn export ../public/fslazywindow/template binTreeViz`
- Edit CMakeLists.txt and change the TARGET\_NAME as:  
`set(TARGET_NAME binary_tree_visualizer)`
- Add ysbitmapfont in LIB\_DEPENDENCY as:  
`set(LIB_DEPENDENCY fslazywindow ysbitmapfont)`
- Cut out the binary-tree class and member functions to a header file called bintree.h
- Find keyword “set(HEADERS” in CMakeLists.txt
- Add bintree.h in CMakeLists.txt as (Optional):  
`set(HEADERS  
 ${platform_HEADERS}  
 bintree.h  
)`
- Run CMake and test-build binary\_tree\_visualizer



## Change to the FsLazyWindowApplication

- Add following headers at the beginning of main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ysglfontdata.h>
#include "bintree.h"
```

- Add a member variable in FsLazyWindowApplication class as:

```
BinaryTree <int,int> btree;
```

- In Initialize, do the following:

```
    srand((int)time(nullptr));
    for(int i=0; i<50; ++i)
    {
        btree.Add(rand()%100,0);
    }
```

- Add a member-function declaration:

```
void Draw(
    const BinaryTree <int,int>::Node *node,
    int x0,int x1,int y,int yStep) const;
```

# Change to the FsLazyWindowApplication

- Draw function must look like:

```
void FsLazyWindowApplication::Draw(const BinaryTree <int,int>::Node *node,int x0,int x1,int y,int yStep) const
{
    if(nullptr!=node)
    {
        int x=(x0+x1)/2;

        glColor3ub(0,255,0);
        glBegin(GL_LINES);
        if(nullptr!=node->left)
        {
            int xNext=(x0+x)/2;
            int yNext=y+yStep;
            glVertex2i(x,y);
            glVertex2i(xNext,yNext);
        }
        if(nullptr!=node->right)
        {
            int xNext=(x+x1)/2;
            int yNext=y+yStep;
            glVertex2i(x,y);
            glVertex2i(xNext,yNext);
        }
        glEnd();

        glColor3ub(0,0,0);
        glRasterPos2d(x-15,y);

        char str[16];
        sprintf(str,"%d",node->key);
        YsGIDrawFontBitmap10x14(str);

        Draw(node->left,x0,x,y+yStep,yStep);
        Draw(node->right,x,x1,y+yStep,yStep);
    }
}
```

Modify the original Draw function as:

```
void FsLazyWindowApplication::Draw(void)
{
    int wid,hei;
    FsGetWindowSize(wid,hei);

    glViewport(0,0,wid,hei);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0,(float)wid-1,(float)hei-1,0,-1,1);

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    Draw(btree.GetRoot(),0,wid,40,40);
    FsSwapBuffers();
    needRedraw=false;
}
```

Identify on which node the mouse cursor is on.

- Add the following member variable in FsLazyWindowApplication class, and set nullptr in the constructor:

```
const BinaryTree <int,int>::Node *mouseOn;
```

- Add the following function declarations in FsLazyWindowApplication class.

public:

```
const BinaryTree <int,int>::Node *PickedNode(int mx,int my) const;
```

private:

```
const BinaryTree <int,int>::Node *PickedNode(  
    const BinaryTree <int,int>::Node *node,  
    int mx,int my,int x0,int x1,int y,int yStep) const;
```

# Implement PickedNode functions.


```
const BinaryTree <int,int>::Node *FsLazyWindowApplication::PickedNode(int mx,int my) const
{
    int wid,hei;
    FsGetWindowSize(wid,hei);
    return PickedNode(btree.GetRoot(),mx,my,0,wid,40,40);
}

const BinaryTree <int,int>::Node *FsLazyWindowApplication::PickedNode(
    const BinaryTree <int,int>::Node *node,
    int mx,int my,int x0,int x1,int y,int yStep) const
{
    if(nullptr!=node)
    {
        int x=(x0+x1)/2;
        if(x-15<=mx && mx<=x+15 && y-14<=my && my<=y)
        {
            return node;
        }
        auto picked=PickedNode(node->left,mx,my,x0,x,y+yStep,yStep);
        if(nullptr!=picked)
        {
            return picked;
        }
        picked=PickedNode(node->right,mx,my,x,x1,y+yStep,yStep);
        if(nullptr!=picked)
        {
            return picked;
        }
    }
    return nullptr;
}
```


Start the same recursion  
as Draw.



If it is the node on which the mouse  
cursor is on, return it.



If not, continue search, and  
stop as soon as the node is  
found.



If not found, return nullptr.

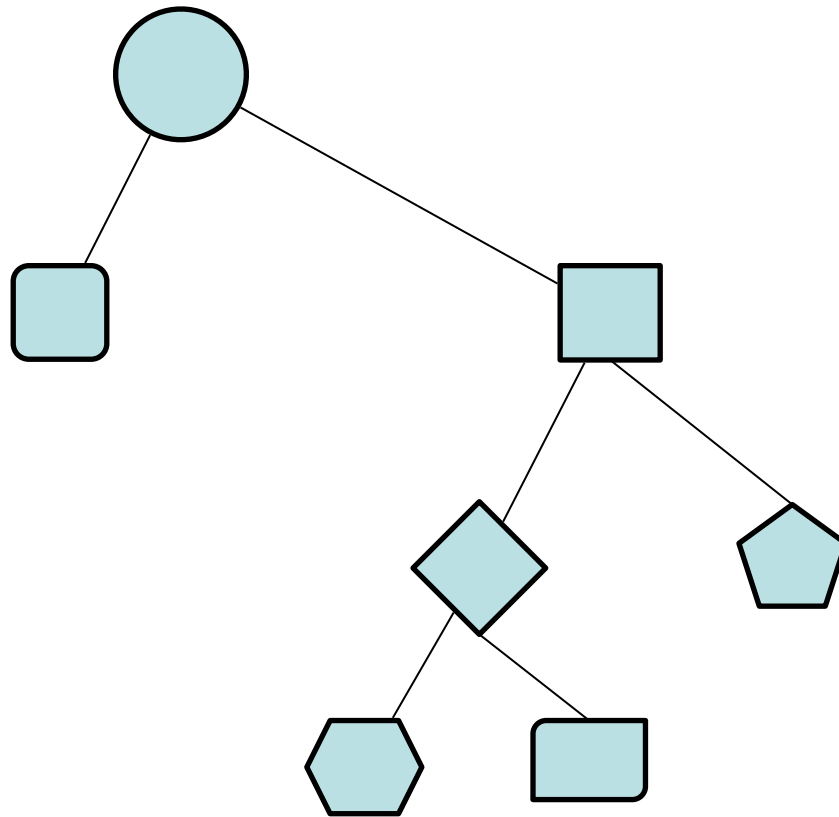
Add the following three lines in Interval so that the node where mouse cursor is on is cached.

```
int lb,mb,rb,mx,my;  
FsGetMouseEvent(lb,mb,rb,mx,my);  
mouseOn=PickedNode(mx,my);
```

Now you can select a node.

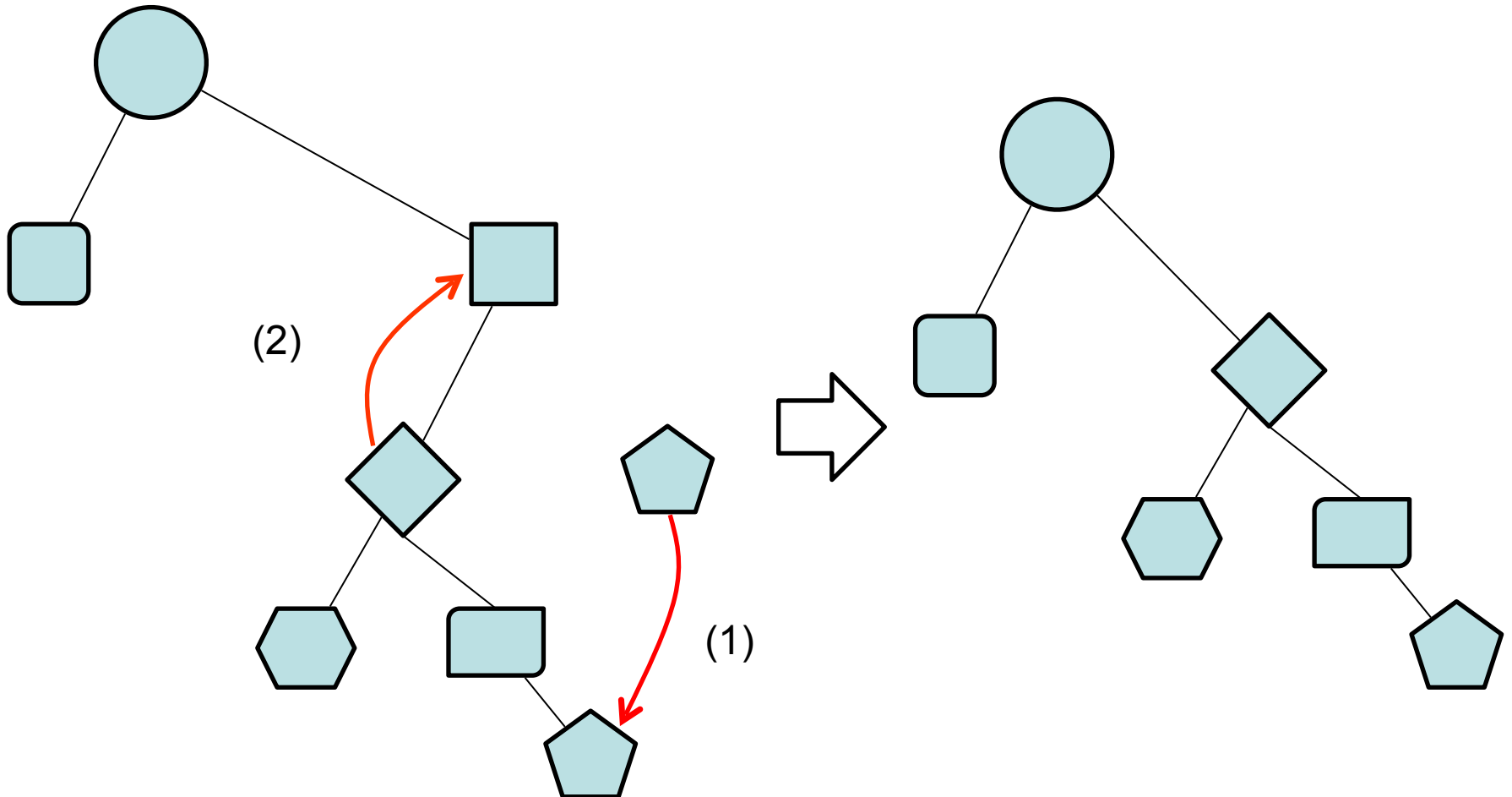
## Deleting a binary-tree node

- Problem: Want to delete one node from the binary tree without breaking the order in the tree.



## Deleting a binary-tree node

- Sloppy method. (1) Connecting the right sub-tree to the right-most node of the left-sub-tree, and then (2) put left sub-tree in position for the node to be deleted.



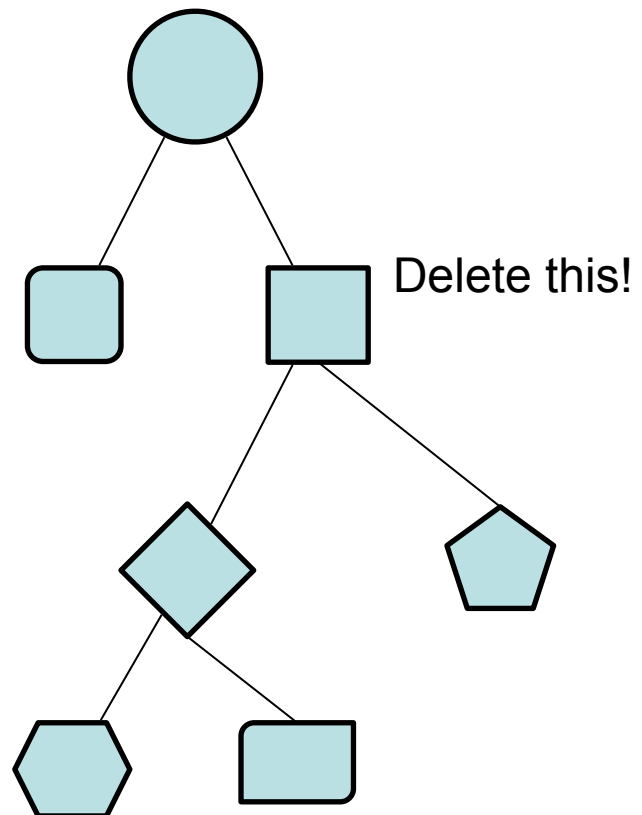


## Problem of the sloppy method

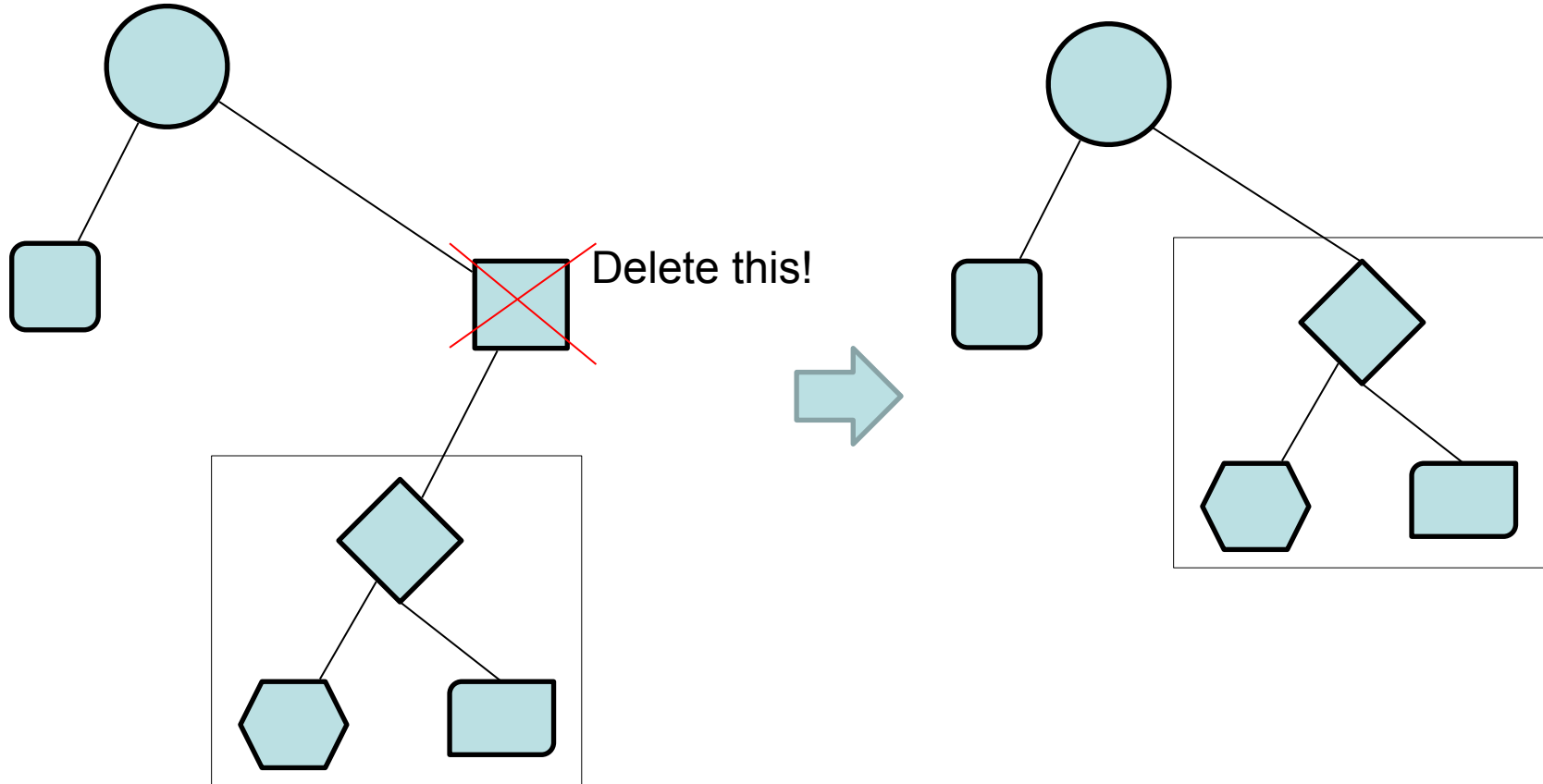
- The height of the tree may increase after deletion.
- We are deleting a node. Want to keep the tree height same or even shorter.

## Deleting a binary-tree node

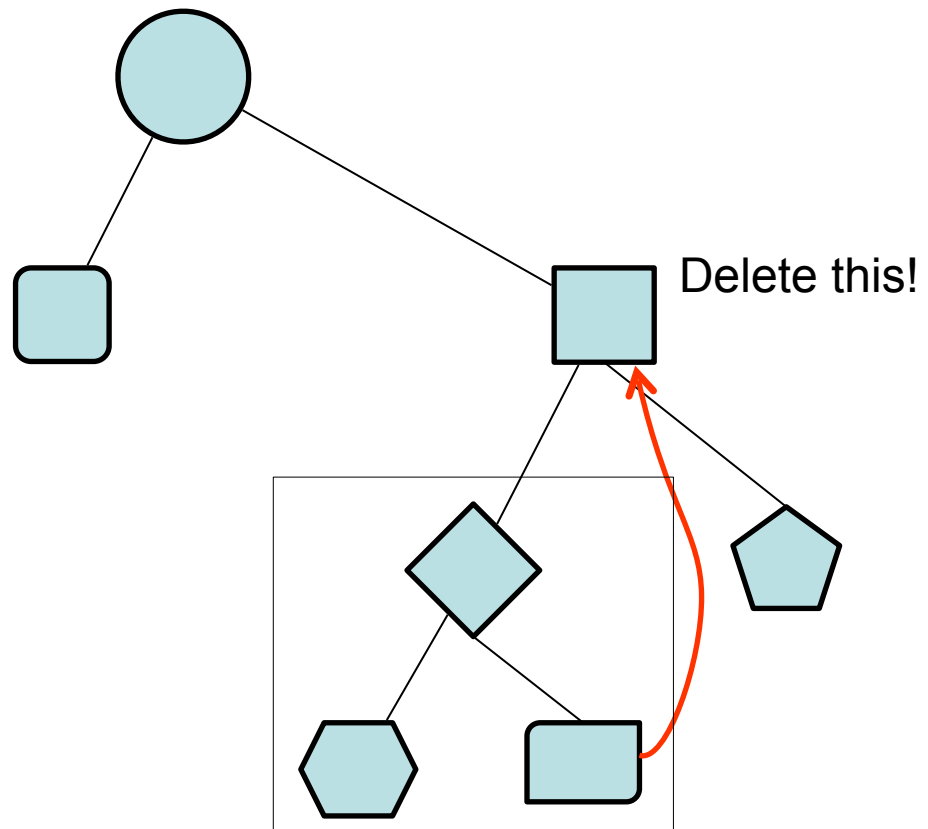
- Good method.
- Want to delete a square in the binary-tree below.
- Easiest case: The node to be deleted has null left or right.
- General case.



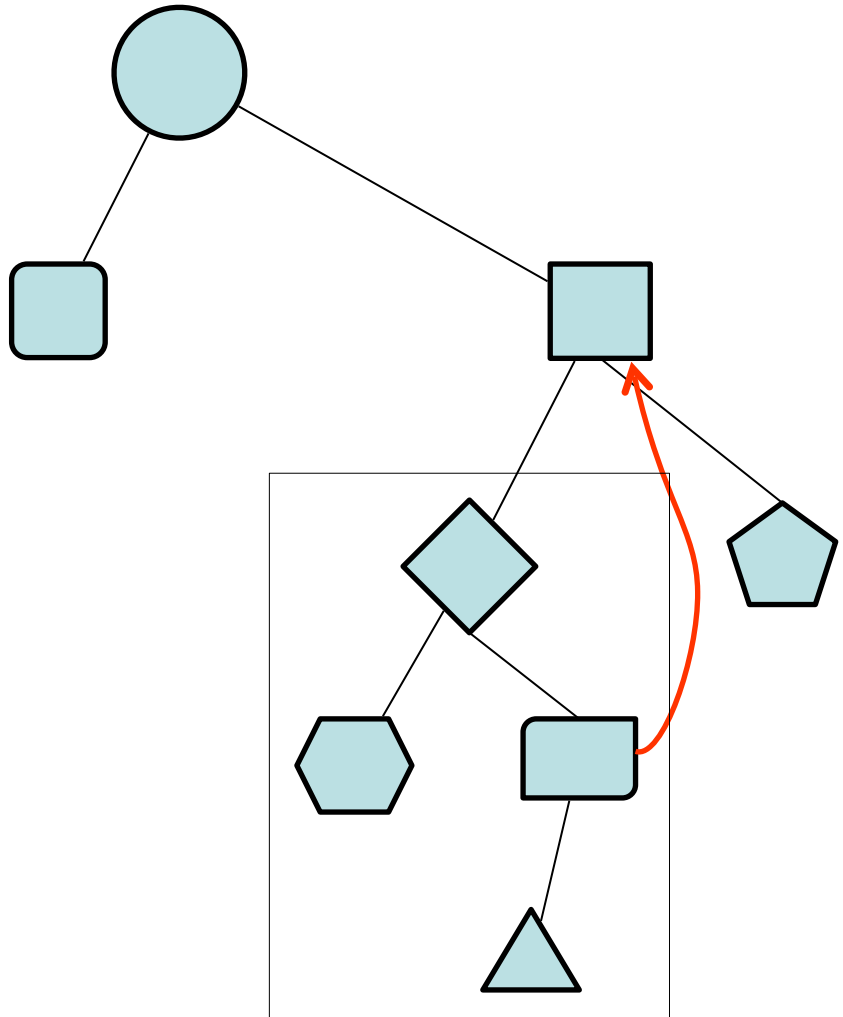
- Easiest case: If the node to be deleted has null left or right, then...
- Put non-null leaf in place for the deleted node.



- General case: If both left and right of the node to be deleted are non-null, then...
- The right-most of the left sub-tree need to take position for the deleted node. (Or, left-most of the right-sub-tree. It is symmetric.)



- The right-most of the left sub-tree may have its left node.
- So, first detach the right-most of the left sub-tree safely, and then replace.



- Add SimpleDetach, SimpleAttach, and then Delete functions in the BinaryTree class.
- SimpleDetach needs to return the position (left, right, or root) so that another node can be re-attached to where the node is detached.

- Add member-function declarations:

public:

```
class Position
```

```
{
```

```
public:
```

```
    Node *node;
```

```
    int position;
```

```
};
```

```
void Delete(const Node *toDel);
```

private:

```
    Position SimpleDetach(Node *toDetach);
```

```
    void SimpleAttach(Node *toAttach, Position pos);
```

- Implementations

```
template <class KeyClass,class ValueClass>
typename BinaryTree<KeyClass,ValueClass>::Position
BinaryTree<KeyClass,ValueClass>::SimpleDetach(Node *toDetach)
{
    Position pos;
    if(nullptr!=toDetach->parent)
    {
        if(toDetach->parent->left==toDetach)
        {
            pos.node=toDetach->parent;
            pos.position=POSITION_LEFT;
            toDetach->parent->left=nullptr;
            toDetach->parent=nullptr;
        }
        else // if(toDetach->parent->right==toDetach)
        {
            pos.node=toDetach->parent;
            pos.position=POSITION_RIGHT;
            toDetach->parent->right=nullptr;
            toDetach->parent=nullptr;
        }
    }
    else
    {
        pos.node=nullptr;
        pos.position=POSITION_ROOT;
        rootNode=nullptr;
    }
    return pos;
}
```



- Implementations

```
template <class KeyClass,class ValueClass>
void BinaryTree<KeyClass,ValueClass>::SimpleAttach(Node *toAttach,Position pos)
{
    if(POSITION_ROOT==pos.position || nullptr==pos.node)
    {
        rootNode=toAttach;
        toAttach->parent=nullptr;
    }
    else if(POSITION_LEFT==pos.position)
    {
        pos.node->left=toAttach;
        toAttach->parent=pos.node;
    }
    else // if(POSITION_RIGHT==position)
    {
        pos.node->right=toAttach;
        toAttach->parent=pos.node;
    }
}
```

## • Implementations

```
template <class KeyClass,class ValueClass>
```

```
void BinaryTree<KeyClass, ValueClass>::
```

```
    Delete(const Node *cToDel)
```

```
{
```

```
    Node *toDel=(Node *)cToDel;
```

```
    // Easy case?
```

```
    if(nullptr==toDel->left ||
```

```
        nullptr==toDel->right)
```

```
{
```

```
    auto pos=SimpleDetach(toDel);
```

```
    if(nullptr!=toDel->left)
```

```
    {
```

```
        SimpleAttach(toDel->left,pos);
```

```
    }
```

```
    else if(nullptr!=toDel->right)
```

```
    {
```

```
        SimpleAttach(toDel->right,pos);
```

```
    }
```

```
    delete toDel;
```

```
}
```

```
else
```

```
{
```

```
    // Find right-most of the left sub-tree.
```

```
    Node *rightMostOfLeft=toDel->left;
```

```
    while(nullptr!=rightMostOfLeft->right)
```

```
    {
```

```
        rightMostOfLeft=rightMostOfLeft->right;
```

```
    }
```

```
{
```

```
    auto pos=SimpleDetach(rightMostOfLeft);
```

```
    if(nullptr!=rightMostOfLeft->left)
```

```
    {
```

```
        SimpleAttach(rightMostOfLeft->left,pos);
```

```
    }
```

```
}
```

```
{
```

```
    auto pos=SimpleDetach(toDel);
```

```
    SimpleAttach(rightMostOfLeft,pos);
```

```
    rightMostOfLeft->right=toDel->right;
```

```
    if(nullptr!=rightMostOfLeft->right)
```

```
    {
```

```
        rightMostOfLeft->right->parent=rightMostOfLeft;
```

```
    }
```

```
    rightMostOfLeft->left=toDel->left;
```

```
    if(nullptr!=rightMostOfLeft->left)
```

```
    {
```

```
        rightMostOfLeft->left->parent=rightMostOfLeft;
```

```
    }
```

```
}
```

```
    delete toDel;
```

```
}
```

```
}
```

- Test code in the Interval function.

```
void FsLazyWindowApplication::Interval(void)
{
    int lb,mb,rb,mx,my;
    FsGetMouseEvent(lb,mb,rb,mx,my);
    mouseOn=PickedNode(mx,my);

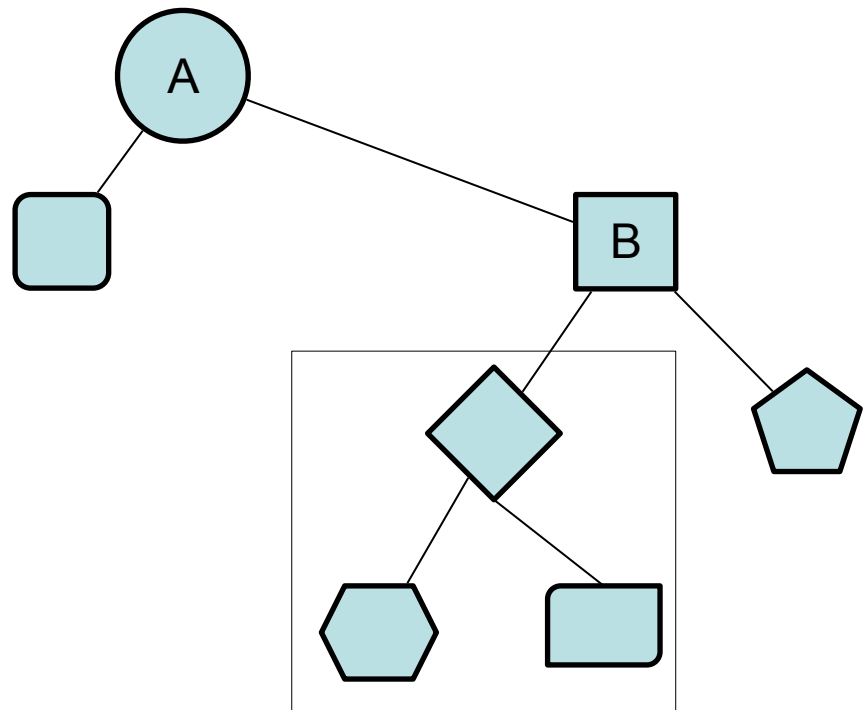
    auto key=FsInkey();
    if(FSKEY_ESC==key)
    {
        SetMustTerminate(true);
    }
    if(FSKEY_D==key && nullptr!=mouseOn)
    {
        btree.Delete(mouseOn);
    }

    needRedraw=true;
}
```

Can delete a node by D key.

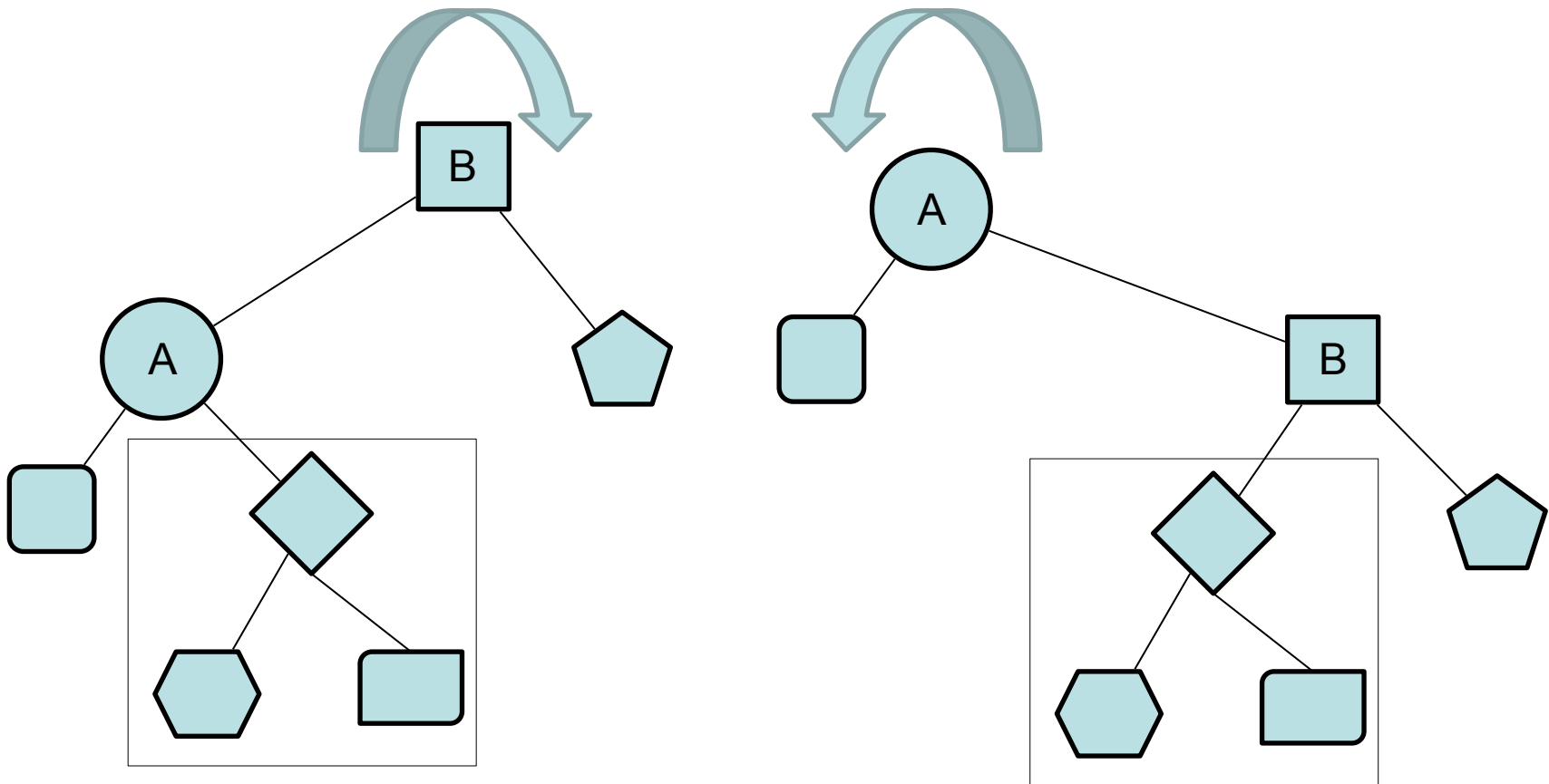
## Tree rotation.

- The left sub-tree of node B is between nodes A and B. I.e., every node in the sub-tree's key must be between the keys of A and B. Therefore,



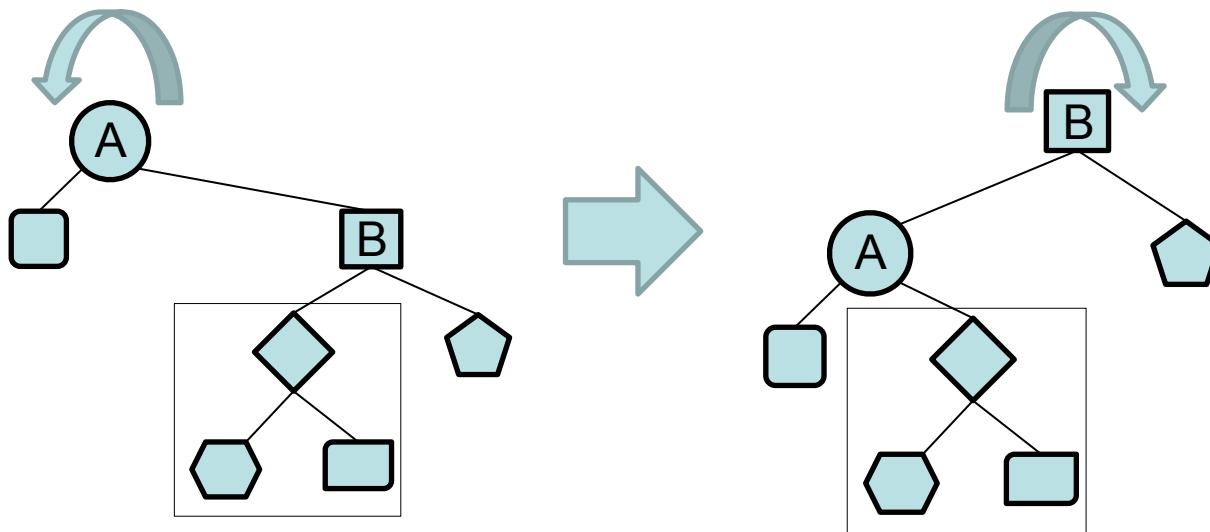
## Tree rotation.

- Two trees below are equivalent.
- The node can be 'rotated' without breaking the node order.



## Left rotation

- Node A can be rotated left only if it has the right node.
  - Detach A and remember the position.
  - Connect left-sub-node of B to the right of A.
  - Connect A to the left of B.
  - Put B in place for A.



## Problem Set 1: Re-balancing a Binary Tree

1. Write your own RotateRight function. It is symmetric with RotateLeft. Should be easy.
2. Implement a tree-rebalancing function. Tree re-balancing can be done with three functions.
  - void TreeToVine(void);
  - void Compress(int n);
  - void VineToTree(void);

The algorithm is based on:

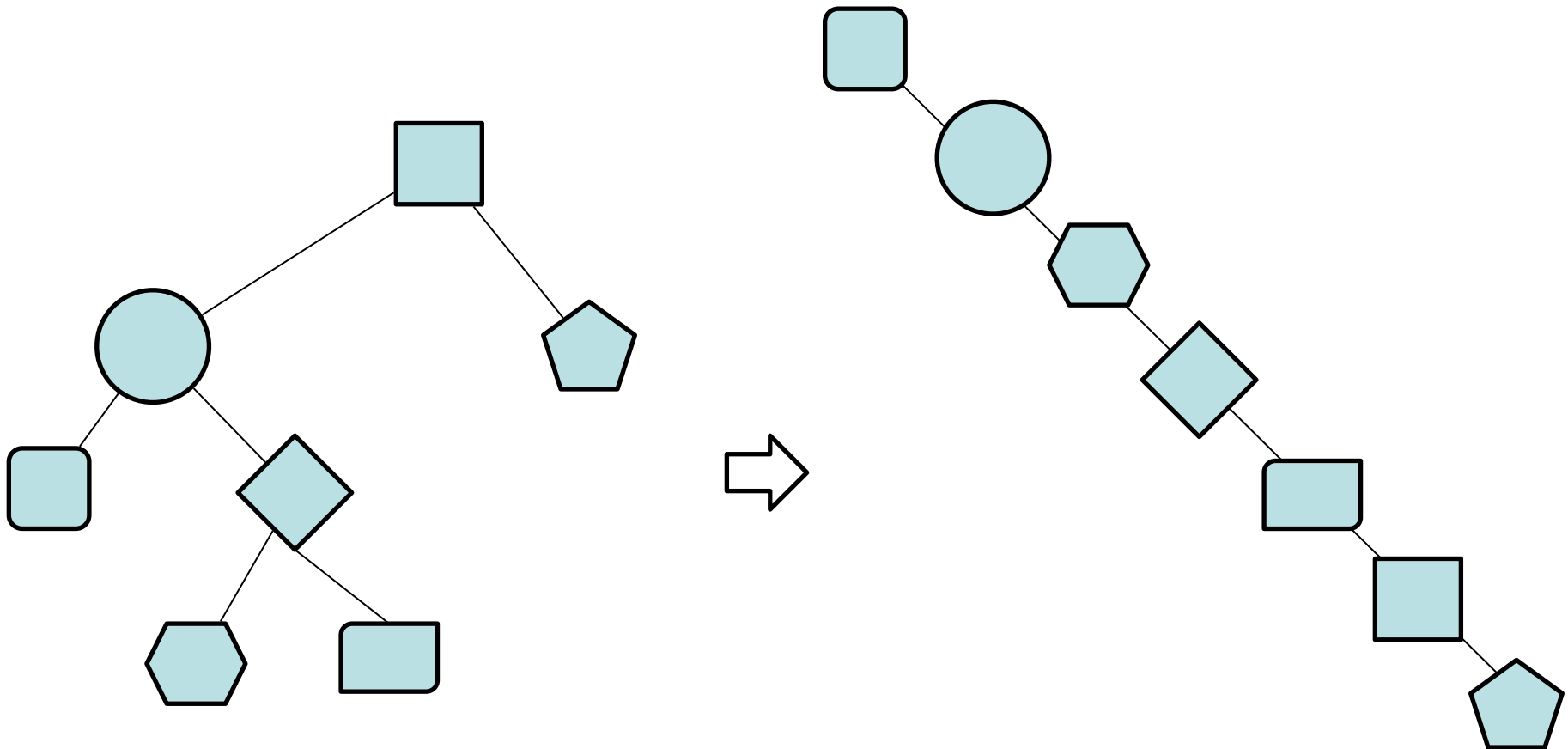
Quentin F. Stout and Bette L. Warren, “*Tree Rebalancing in Optimal Time and Space*,” Communications of the ACM, September 1986, Volume 29, Number 9, pp. 902-908

<http://web.eecs.umich.edu/~qstout/pap/CACM86.pdf>

[https://en.wikipedia.org/wiki/Day%E2%80%93Stout%E2%80%93Warren\\_algorithm](https://en.wikipedia.org/wiki/Day%E2%80%93Stout%E2%80%93Warren_algorithm)

# Tree to Vine

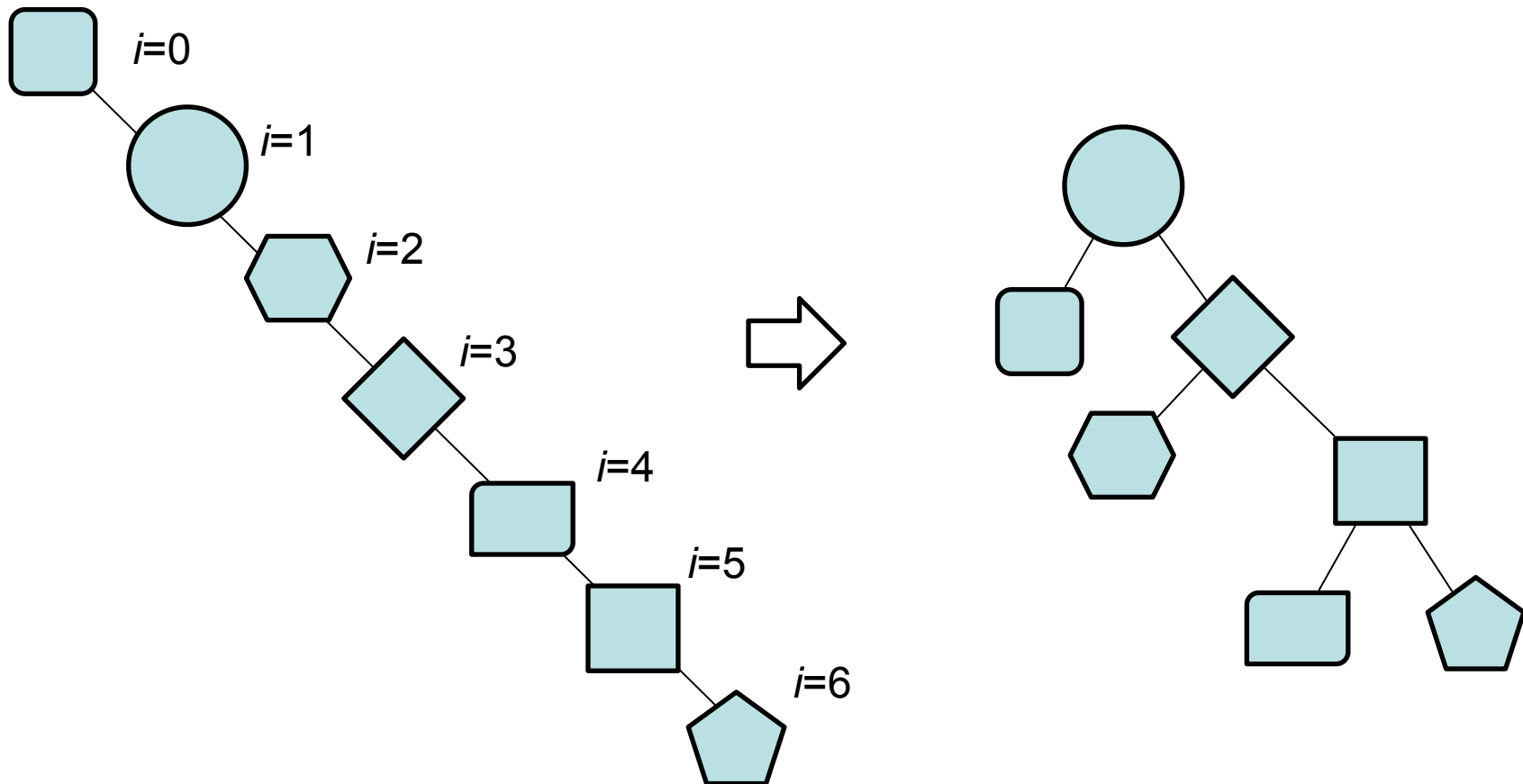
- Apply a sequence of right rotation starting from the root node, until the tree becomes a linear





## Compress

- Takes an input parameter  $N$ . Vine node  $i$  (zero-based) is the  $i$ th node connected from the root by right pointer.
- Apply left rotation to  $i=0, i=2, \dots, i=2*(N-1)$ .

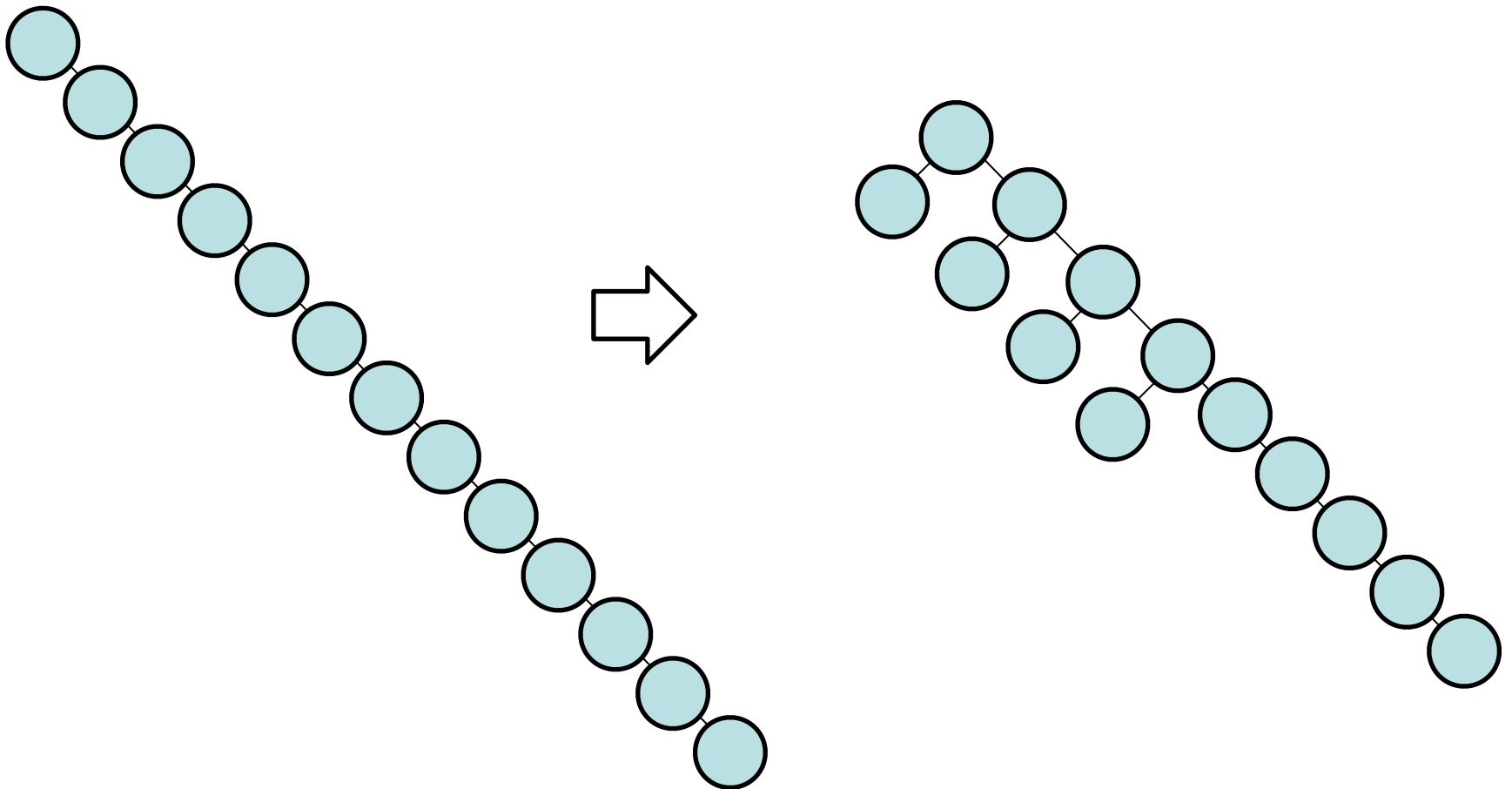


## Vine to Tree

- Apply a sequence of compression to convert a vine to a balanced tree.

```
sz=(node count)
lc=sz+1-2int(log2(sz+1))
Compress(lc)
sz=sz-lc
while(1<sz)
{
    Compress(sz/2)
    sz/=2
}
```

- First compression will make some leaves on the left of the vine nodes. These left nodes will become the deepest nodes in the end.



- Then the subsequent left rotations will make the tree perfectly balanced with deepest nodes on the left

