درختان ۲

Binary Tree Iterator

- کلاسی تعریف کنید که پیمایش را به صورت زیر پیاده کند:
  - از ریشه شروع کند.
  - به نود بعدی برود.
  - داده را ببیند.
  - بطور دلخواه توقف کند.
- باید بصورت تکراری پیاده شود تا قادر به توقف دلخواه باشد.
  - از پشته استفاده می کنیم.

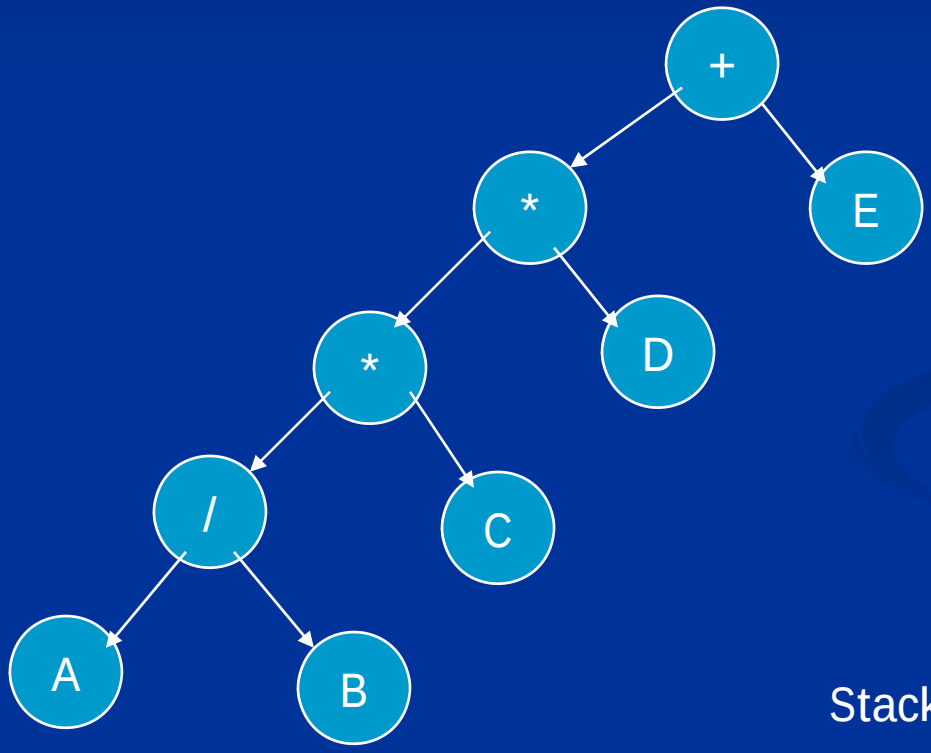
InorderIterator Class

```
class InorderIterator
{
   public:
        InorderIterator(BinaryTree& inputTree):
        tree(inputTree) { CurrentNode = tree.root;};
        char* Next();
   private:
        const BinaryTree& tree;
        Stack<BinaryTreeNode*> stack;
        BinaryTreeNode* currentNode;
}
```

InorderIterator Class

```
char *InorderIterator::Next()
{
    while (currentNode) {
          stack.Add(currentNode);
          currentNode = currentNode->leftChild; }
    if (! stack.isEmpty()) {
          currentNode = *(stack.Delete(currentNode));
          char& temp = &(currentNode->data);
          currentNode = currentNode->rightChild;
          return &temp;
    }
    else return 0; // traversed whole tree
}
```

InorderIterator Validation



A

/ / /

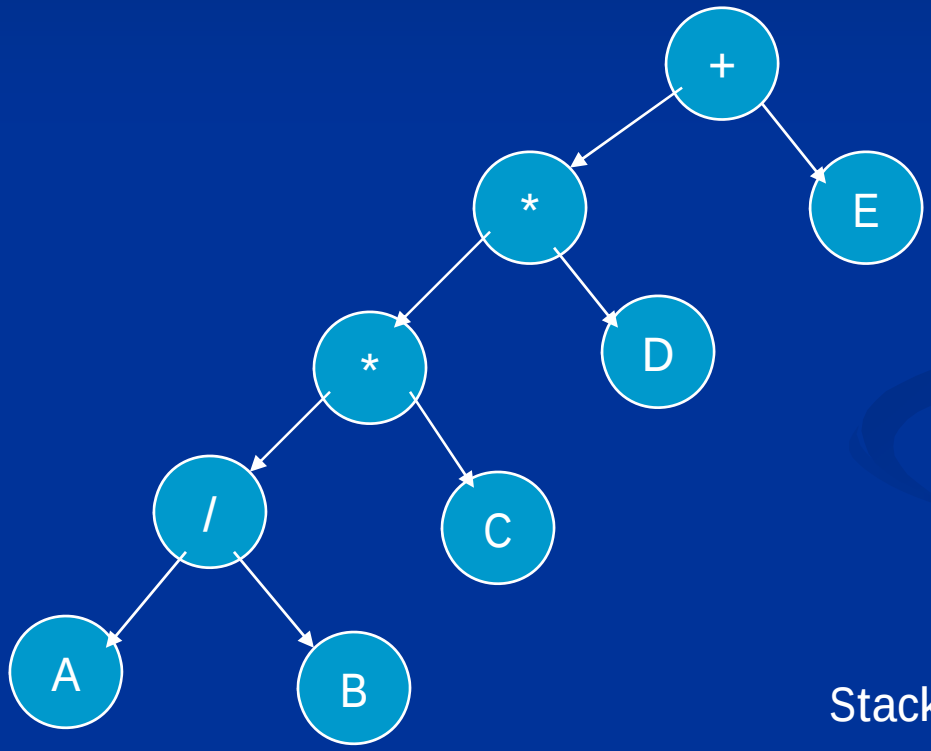
* * * *

* * * * *

Stack: + + + + + +

Output: A

InorderIterator Validation



Stack:

/		B
*	*	*
*	*	*
+	+	+

Output: A / [ETC...]

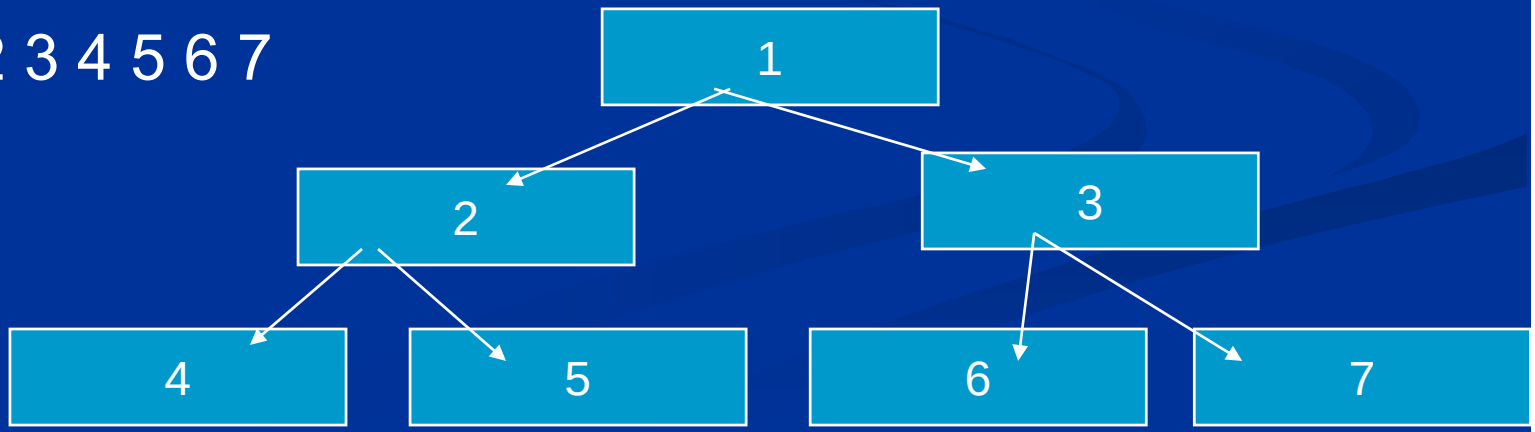
InorderIterator Analysis

- پیچیدگی زمانی:
- n نود داریم.
- هر نود یکبار به پشته اضافه می گردد.
  - هنگام رفتن به پایین
- هر نود یکبار از پشته حذف می گردد.
  - هنگام دیدن نود در مسیر برگشت
- پیچیدگی زمانی: O(n)
- پیچیدگی مکانی: O(height)
  - اگر درخت بالانس باشد: $O(\log_2 n)$
  - اگر درخت خطی باشد: O(n)

Level-Order Traversal

- قوانین پیمایش سطحی:
  - بجای پایین رفتن در سطح حرکت کن.
  - بعد از دیدن تمام نودهای یک سطح به سطح بعدی برو.

1 2 3 4 5 6 7



Level Order Traversal

- پیمایش سطحی به پیمایش اول-سطح نیز معروف است.
- پیمایشهای قبلی از پشته استفاده کردند:
  - البته در روش تکراری به صورت صریح
  - و در روش بازگشتی به صورت ضمنی
- برای انجام پیمایش اول-سطح به یک صف به جای پشته نیازمندیم.

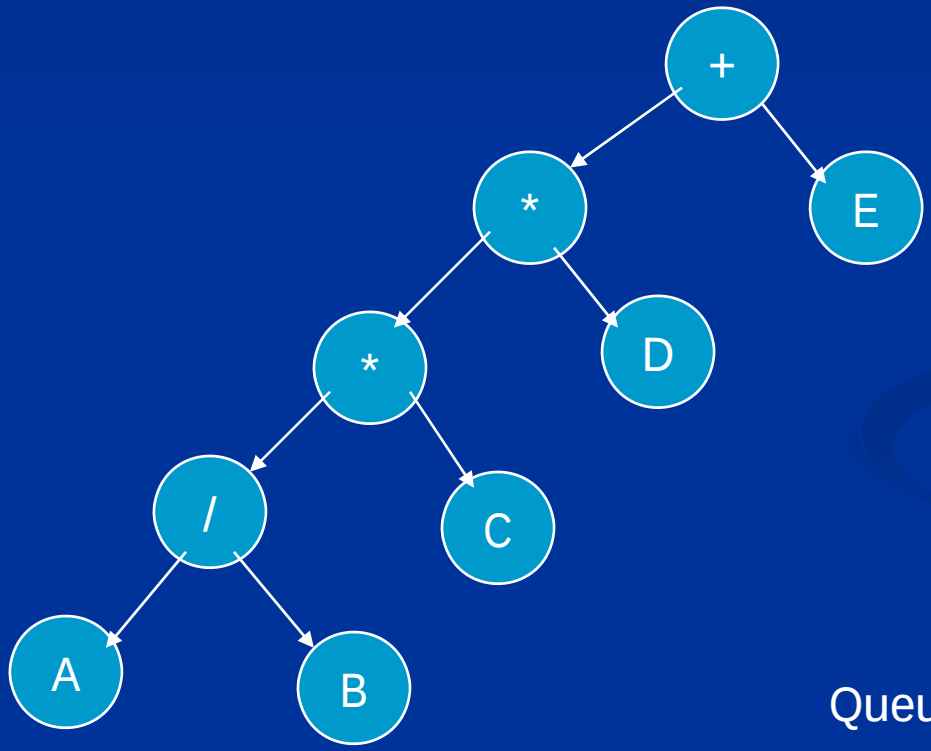
Level Order Traversal

- ریشه را به صف اضافه کنید.
- برای هر نود موجود در صف:
  - نود را بینید
  - فرزند سمت چپ نود را به صف اضافه کنید.
  - سپس فرزند سمت راست نود را به صف اضافه کنید.

Level Order Traversal

```
void BinaryTree::LevelOrder()
{
    Queue<BinaryTreeNode*> queue;
    BinaryTreeNode* currentNode = root;
    while (currentNode) {
        cout << currentNode->data;
        if (currentNode->leftChild)
    queue.Add(currentNode->leftChild);
        if (currentNode->rightChild)
    queue.Add(currentNode->rightChild;
        currentNode = *(queue.Delete(currentNode));
}
```

LevelOrder Validation



Queue: * E | E * D | * D | D / C

Output: + * E *

Binary Tree Operations

- می خواهیم کلاس binary tree را طوری توسعه دهیم که از متدهای زیر پشتیبانی کند:
  - جز سازنده کپی (شبیه عملگر =)
  - عملگر == (ساخت عملگر =! از روی == ساده است).

Binary Tree Operations

- جز سازنده کپی:
  - هدف: ساخت یک درخت جدید که شبیه یک درخت موجود باشد.
  - راه حل: درخت را پیمایش می کنیم و از هر نود که گذشتیم؛ یک نود به درخت جدید اضافه می کنیم.

Binary Tree Copy Constructor

```
BinaryTree::BinaryTree(const BinaryTree & input)
{ root = copy(input.root); }

BinaryTreeNode* copy(BinaryTreeNode* current)
{
    if (current)
    {
            BinaryTreeNode *temp = new BinaryTreeNode;
            temp->data = current->data;
            temp->leftChild = copy(current->leftChild);
            temp->rightChild = copy(current->rightChild);
            return temp;
    }
    else return 0;
}
```

Binary Tree Equivalence

- عملگر ==:
  - هدف: تشخیص دهیم که آیا دو درخت داده شده کاملا شبیه هستند یا نه؟
  - راه حل: درخت را پیمایش کن !!
    - ببین آیا نودهایشان در جاهای یکسانی هستند؟
    - مقادیر داده نودها با هم برابر هستند؟

Binary Tree Equivalence

```
bool operator==(const BinaryTree & tree1, const BinaryTree &
    tree2)
{ return equal(tree1.root, tree2.root); }

bool equal(BinaryTreeNode* a, BinaryTreeNode* b)
{
    if ((!a) && (!b)) return 1; // both null pointers
    if (a && b && (a->data == b->data) // same data
            && equal(a->leftChild,b->leftChild) // same left
            && equal(a->rightChild, b->rightChild) // same right
                    return 1;
    return 0;
}
```

Binary Trees as Tools: Exploiting Traversal

- مساله صدق پذیری:
  - یک عبارت بولی داریم و می خواهیم ببینیم آیا مجموعه ورودیها در آن عبارت صدق می کند یا خیر؟

(!x1 && x2) => x1 = false, x2 = true

Satisfiability Rules

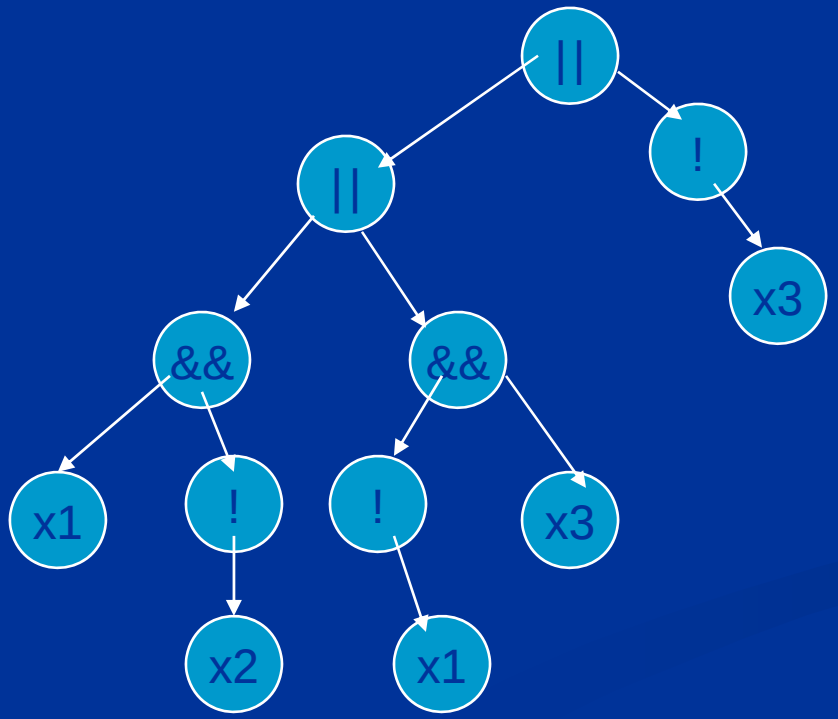
- عبارت:
  - هر متغییر یک عبارت است
  - اگر x و y عبارت هستند پس:
    - x && y ، x || y و !x نیز عبارت هستند.
  - بطور معمول ترتیب عملگر ها به صورت زیر است:

    not > and > or
  - البته پرانتز می تواند ترتیب عملگر ها را به هم بریزد.

Satisfiability Problem

- عبارت را به صورت یک درخت دودویی نمایش دهید.

(x1 && !x2) || (!x1 && x3) || !x3



نودهای انتهایی همه از نوع متغییر هستند.

نودهای غیر انتهایی عملگر هستند.

عملگر ! فقط یک فرزند دارد.

Satisfiability Problem

- درخت را به صورت پس ترتیب (LRV) پیمایش کنید.
- لذا می توانیم پس از ارزیابی فرزندان یک نود، خود نود را ارزیابی کرده و عملگر مربوطه را اعمال کنیم.
  - مقدار هر نود به مقدار فرزندان بستگی دارد.
- باید تمام ترکیبات true/false ورودیها (نودهای انتهایی) را امتحان کنیم.

Satisfiability Problem

- تعریف نود را تغییر می دهیم:
  - Data نوع نود را مشخص می کند (and, or, true, false)
  - Value مقدار نود را مشخص می کند
    - برای نودهای انتهایی true یا false
    - برای نودهای غیر انتهایی وابسته به مقدار فرزندان و نوع عملگر است.

```
class SatNode
{
   friend class SatTree;
   private:
        SatNode *leftChild;
        TypesOfData data;
        bool value;
        SatNode* rightChild;
}
```

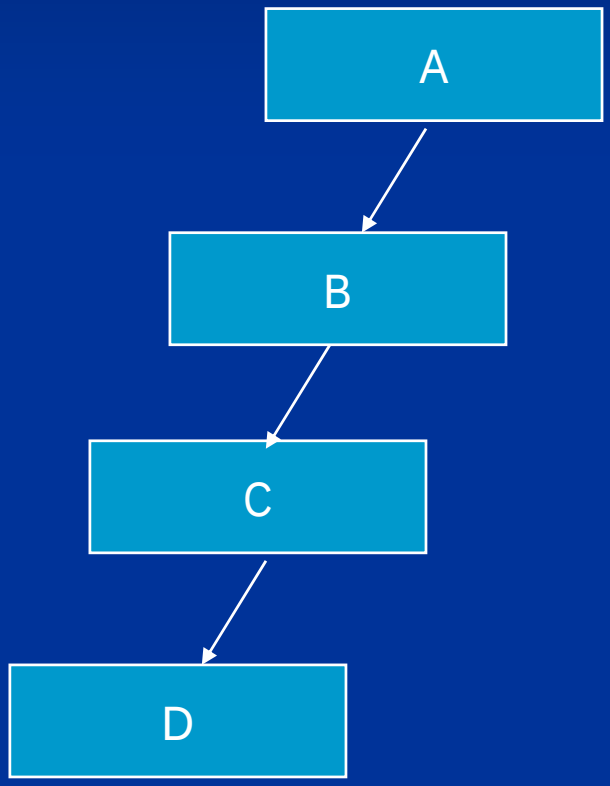
Satisfiability Problem

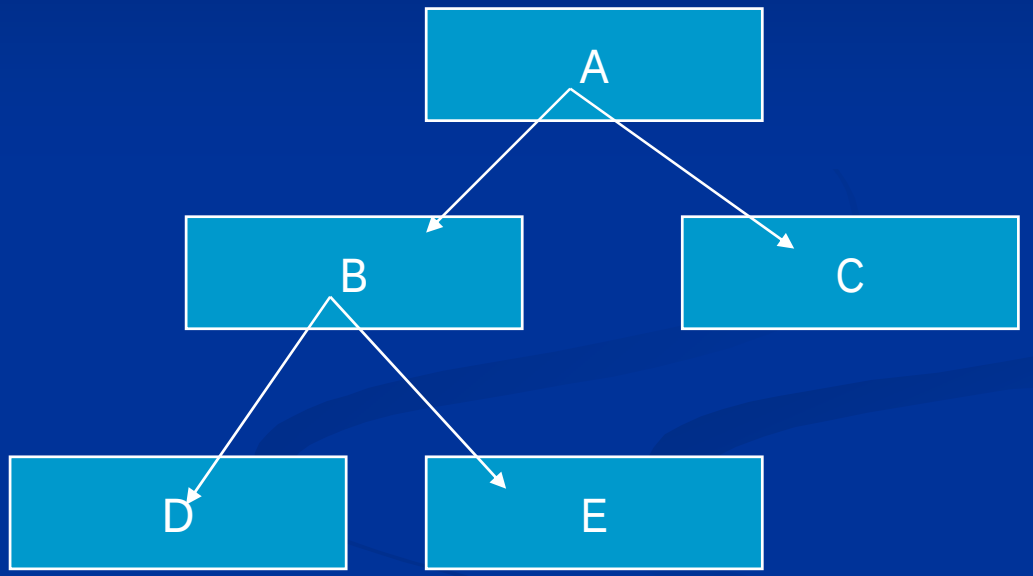
- در سطح اصلی، ما تمام حالتهای ممکن برای ورودیها (نودهای انتهایی) را تنظیم می کنیم.
- سپس درخت را پیمایش کرده؛ و از پایین به سمت ریشه حرکت می کنیم و مقادیر نودها را با اعمال عملگرها به دست می آوریم.

Satisfiability Problem

```
void SatTree::PostOrderEval() {        PostOrderEval(root); }

void SatTree::PostOrderEval(SatNode *s)
{
    if (node)  {
        PostOrderEval(s->leftChild); PostOrderEval(s->rightChild);
        switch (s->data) {
                case LogicalNot: s-> value = !(s->rightChild->value);
    break;
                case LogicalAnd: s->value = s->leftChild->value && s-
    >rightChild->value; break;
                case LogicalOr: s->value = s->leftChild->value || s-
    >rightChild->value; break;
                case LogicalTrue: s->value = 1; break;
                case LogicalFalse: s-value = 0; break; } } }
```

Review Questions



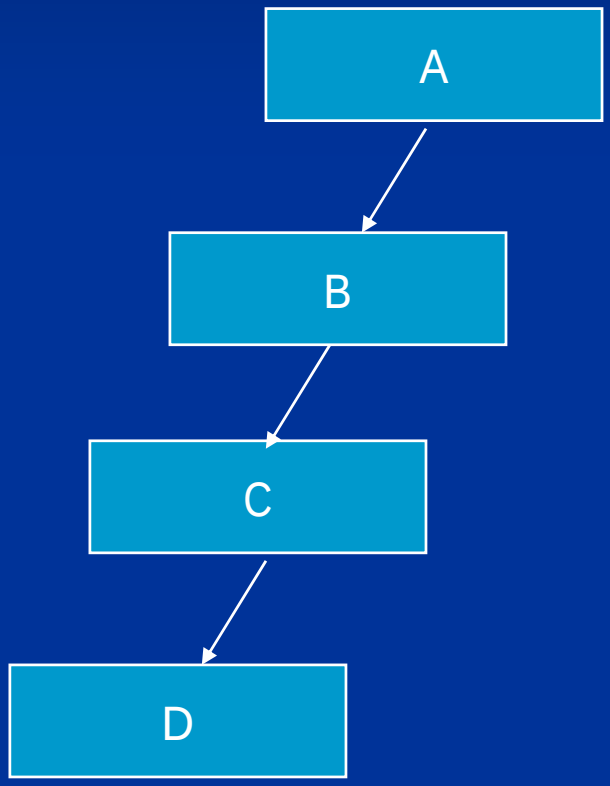


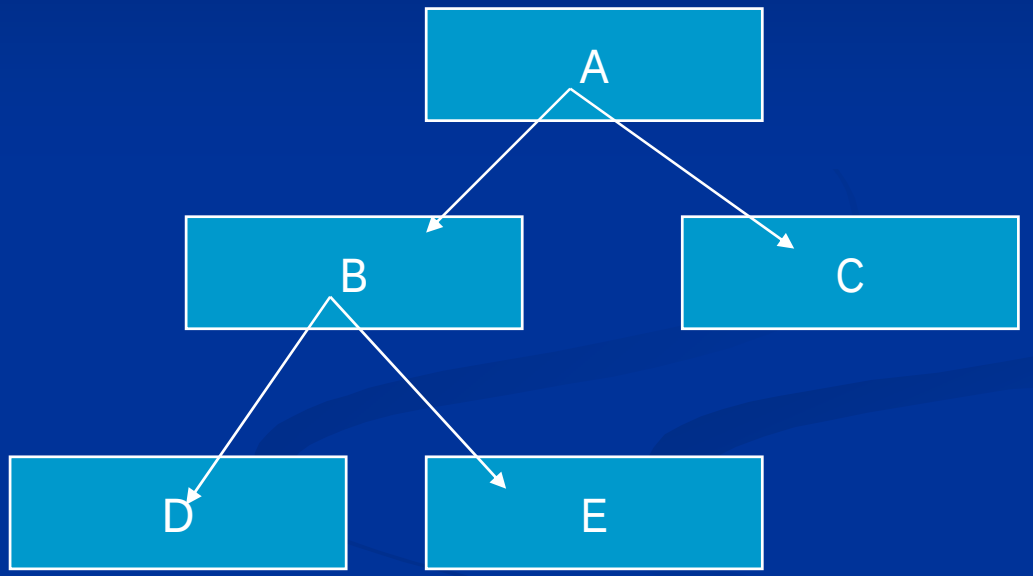
Inorder traversal? (LVR)

D C B A

D B E A C

Review Questions



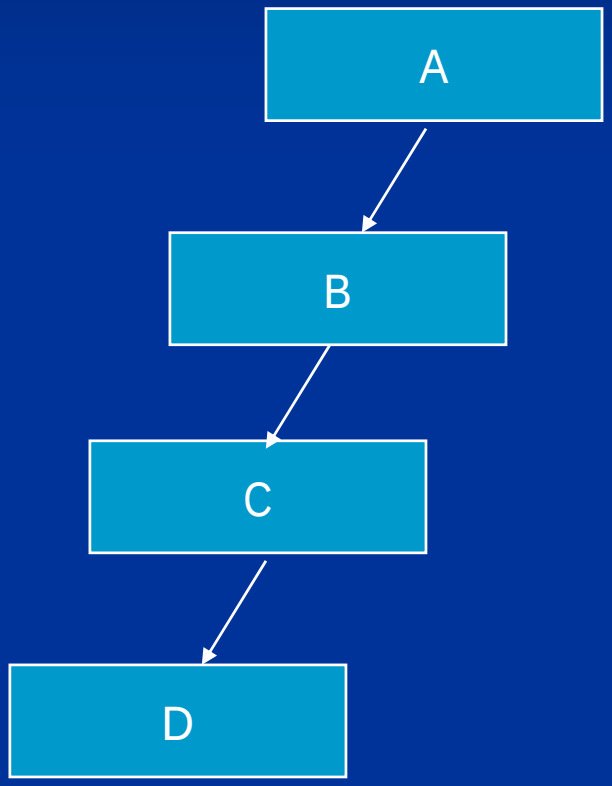


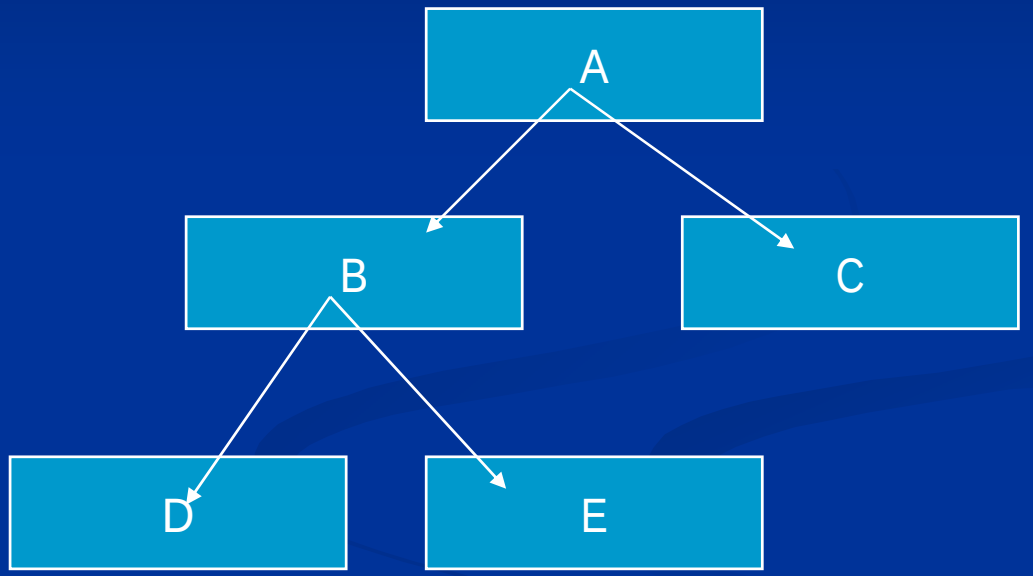
Preorder traversal? (VLR)

A B C D

A B D E C

Review Questions





Postorder traversal? (LRV)

D C B A

D E B C A

Review Questions

- تابعی بنویسید که تعداد نودهای انتهایی را حساب کند:

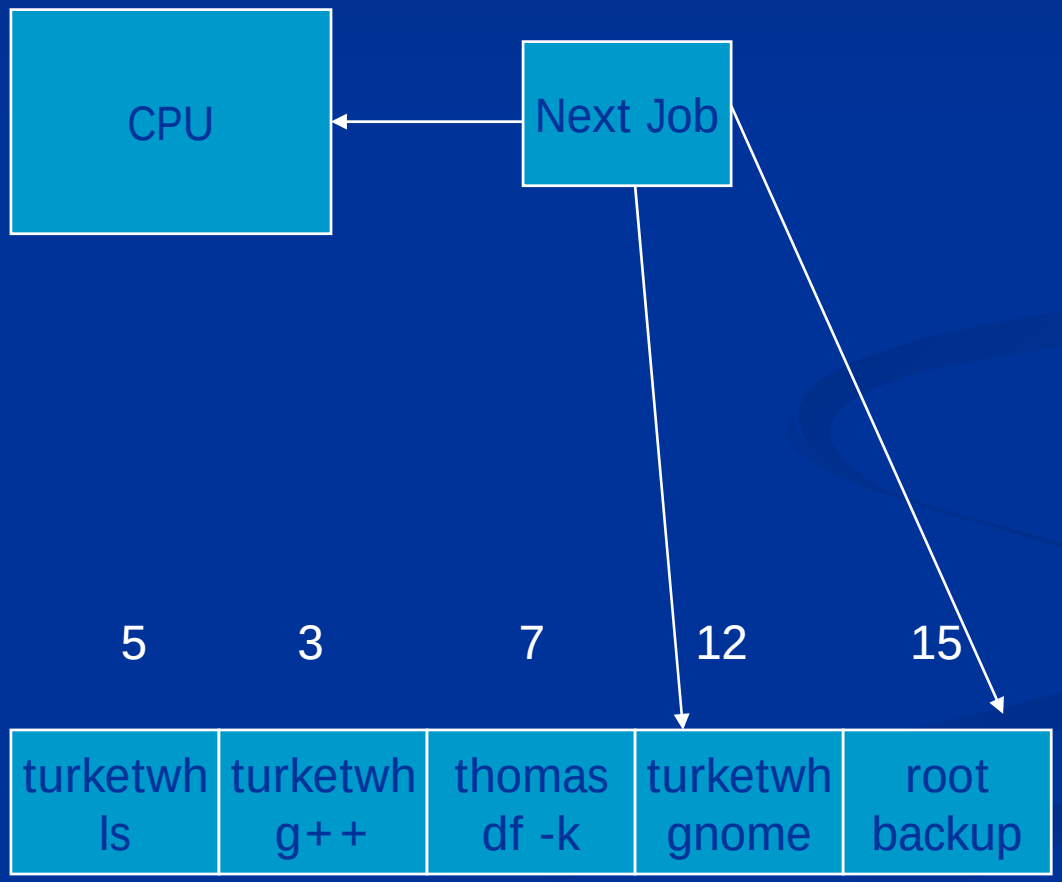
```
int BinaryTree::CountTerminals()
{    return CountTerminals(root); }

int BinaryTree::CountTerminals(BinaryTreeNode* node)
{
 int left = 0;
 int right = 0;
 if ((node->leftChild == 0) && (node->rightChild == 0)) return 1;
 else {
     if (node->leftChild != 0) left = CountTerminals(node->leftChild);
     if (node->rightChild != 0)  right = CountTerminals(node-
    >rightChild);
     return left + right; } }
```

Priority Queues

- تعریف استاندارد صف:
  - به انتها اضافه کن
  - از ابتدا بردار

- تعریف صف اولویت دار:
  - به انتها اضافه کن، بالاترین اولویت را بردار
    - زمانبندی OS
    - سویچینگ بسته ها

Priority Queues – OS Job Scheduling



Priority Queues

- مثال :
  - اجاره دادن ماشین شست و شوی فرش
  - هدف: افزایش تعداد کاربران در زمان محدود
    - ماشین را به خانه های کوچکتر بده (a **min priority** queue)
  - هدف: افزایش درآمد در زمان محدود:
    - ماشین را به کسانی بده که بیشتر پول می دهند. (a **max priority** queue)

Priority Queue

تعريف واسط:

```
template <class Type>
class MaxPriorityQueue
{
   public:
        virtual void Insert (const Element<Type> &
   toInsert) = 0;
        virtual Element<Type>* Delete (Element<Type>
   &) = 0; //(DeleteMax or DeleteMin)
}
```

Priority Queue Implementations

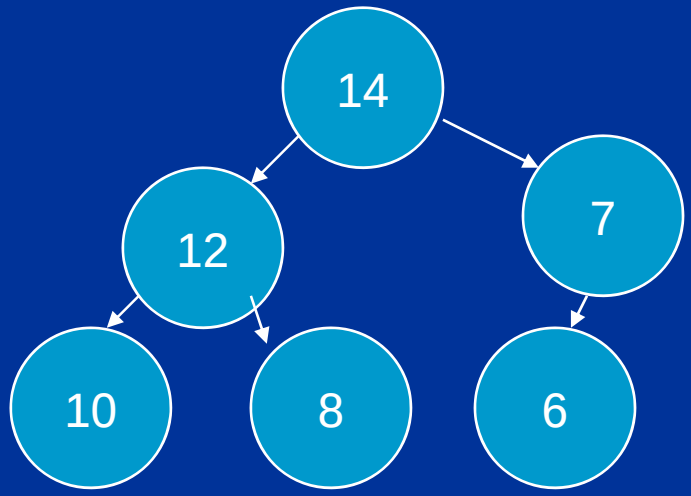
- پیاده سازی لیست پیوندی
- لیست نامرتب:
  - Insert – O(1)
    - اضافه کردن به اول لیست
  - Delete – O(N)
    - جستجوی تمام لیست
- لیست مرتب:
  - Insert – O(N)
    - پیدا کردن جای مناسب در لیست
  - Delete – O(1)
    - حذف از انتها

Heaps

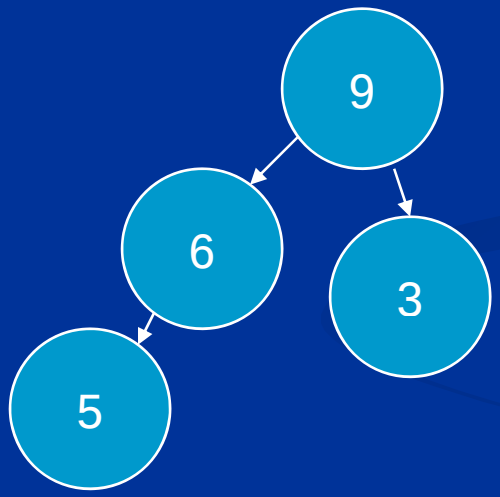
- بهترین راه پیاده سازی صف اولویت دار استفاده از heap است.
- تعریف :heap
  - یک درخت حداکثر درختی است که مقدار هر نود از مقدار داده فرزندان کمتر نباشد. یک heap حداکثر یک درخت کامل و حداکثر است.
    - ریشه بزرگترین است.
  - یک درخت حداقل درختی است که مقدار هر نود از مقدار داده فرزندان بیشتر نباشد. یک heap حداقل یک درخت کامل و حداقل است.
    - ریشه کوچکترین است.

Heap Examples:

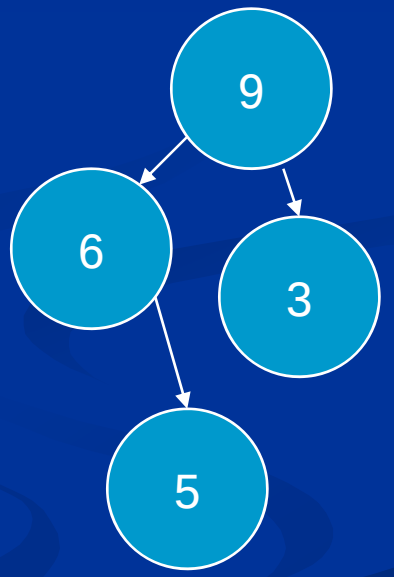
- Heap حداکثر



Complete binary? Yes
Max Tree Property? Yes



Complete binary? Yes
Max Tree Property? Yes

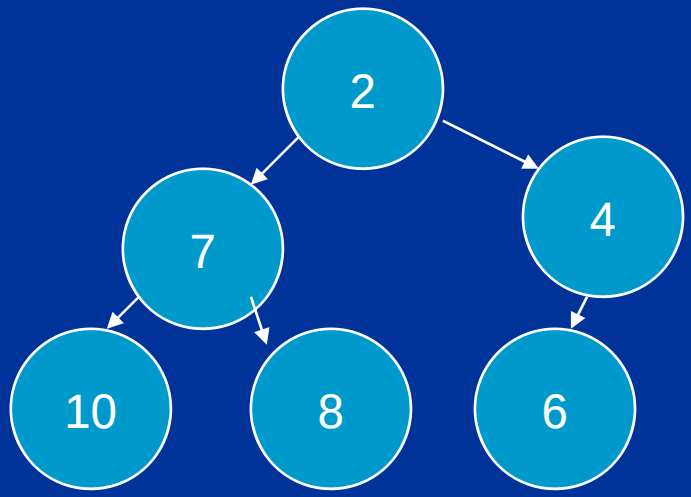


Not A Max Heap!

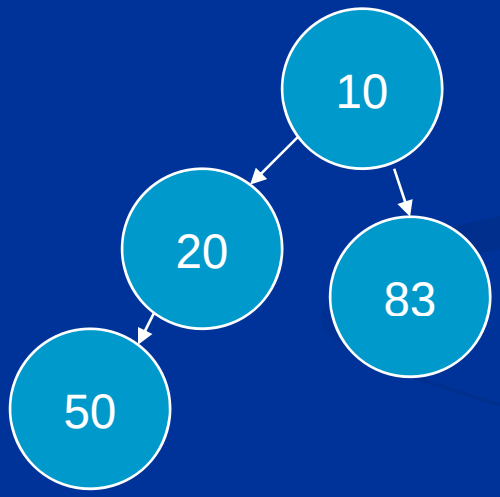
Complete binary? No
Max Tree Property? Yes

Heap Examples:

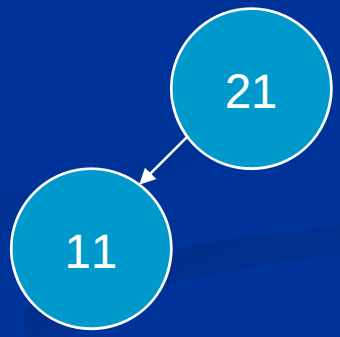
- Heap حداقل:



Complete binary? Yes
Min Tree Property? Yes



Complete binary? Yes
Min Tree Property? Yes



Not A Min Heap!

Complete binary? Yes
Min Tree Property? No

Max Heap Operations

- حداقل واسط مورد نیاز :
  - ایجاد یک heap خالی
  - اضافه کردن یک نود جدید به heap
  - حذف بزرگترین نود از heap

شبیه صف اولویت دار!

Max Heap Definition: Variables

- چون heap یک درخت باینری کامل است، از آرایه برای نمایش آن استفاده می کنیم.

```
class MaxHeap
{
   public:
   private:
        Element<Type> *heap;
        int currentSize;
        int maxSize;
}
```

Max Heap Definition: Methods

```
class MaxHeap
{
   public:
        MaxHeap(int size);
        bool isFull();
        bool isEmpty();
        void insert(Element<KeyType> toInsert);
        Element<KeyType>* delete(KeyType&
   toDelete);
   private:
   // see previous slide
}
```

Max Heap Implementation

سازنده:

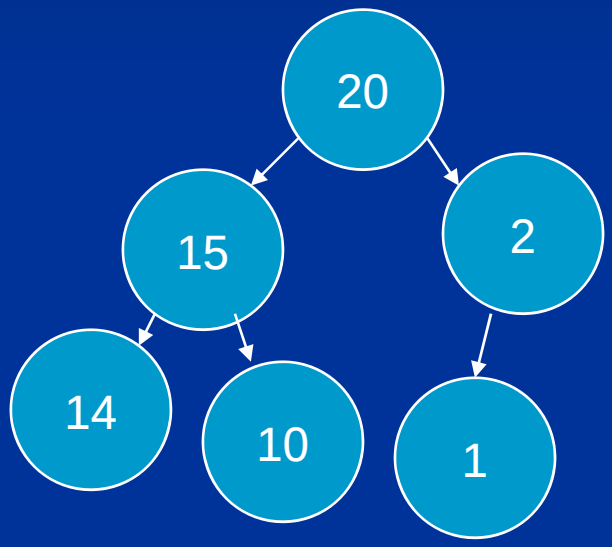
```
template<class KeyType>
MaxHeap<KeyType>::MaxHeap(int size)
{
    maxSize = size;
    currentSize = 0;
    heap = new Element<KeyType>[maxSize+1];
    // heap[0] not used – simplifies binary tree
   // mapping
}
```

isFull, isEmpty()

simple one-line conditional checks

currentSize == maxSize or currentSize == 0

Max Heap: Insertion

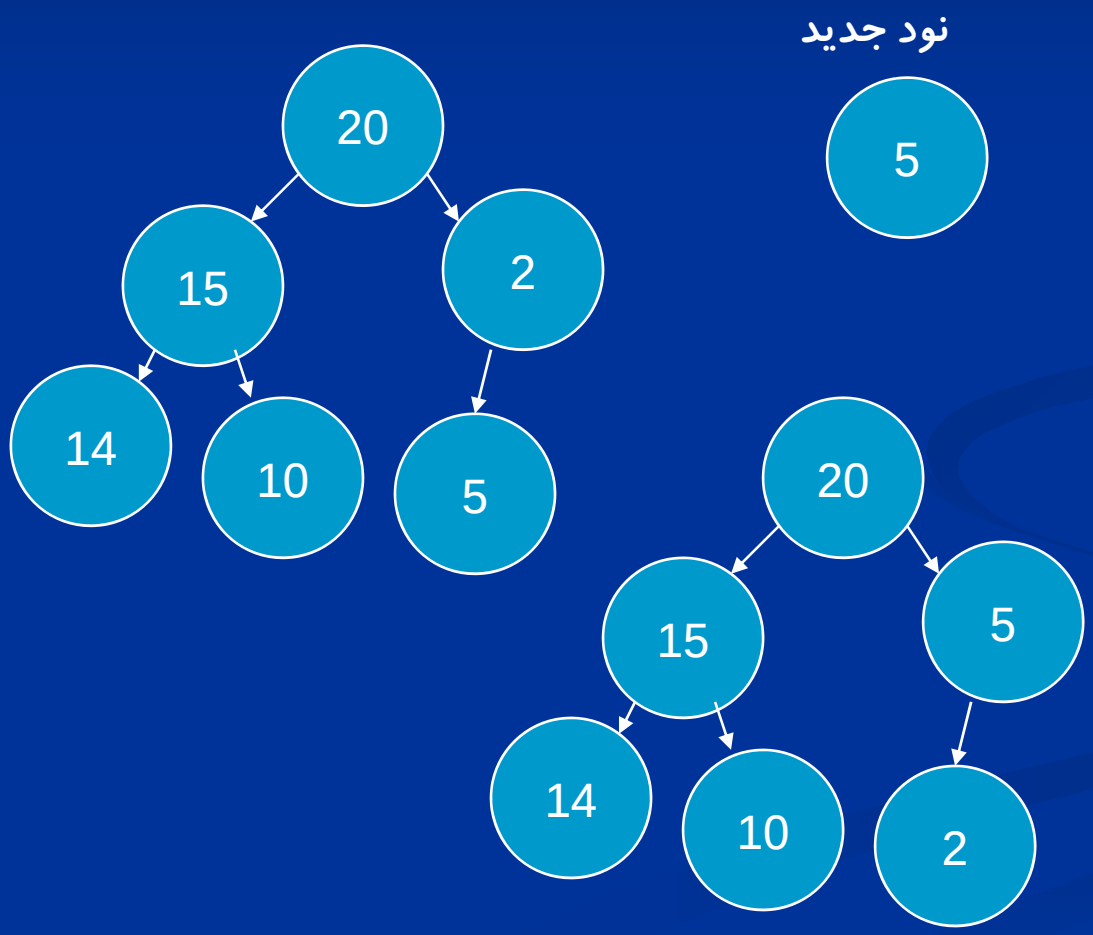


نود جدید

قانون اول:
خاصیت کامل بودن را رعایت کنید. نود را به جای درست اضافه کن

قانون دوم:
خاصیت حداکثر بودن درخت را رعایت کنید.

Max Heap: Insertion



قانون اول:
خاصیت کامل بودن را رعایت کنید. نود را به جای درست اضافه کن

قانون دوم:
خاصیت حداکثر بودن درخت را رعایت کنید. یعنی، تا وقتی که از پدر بزرگتر است، جای نود جدید را با پدر عوض کن

اگر بجای ۵، ۲۵ را اضافه می کردیم باید تا نود ریشه می رفتیم.

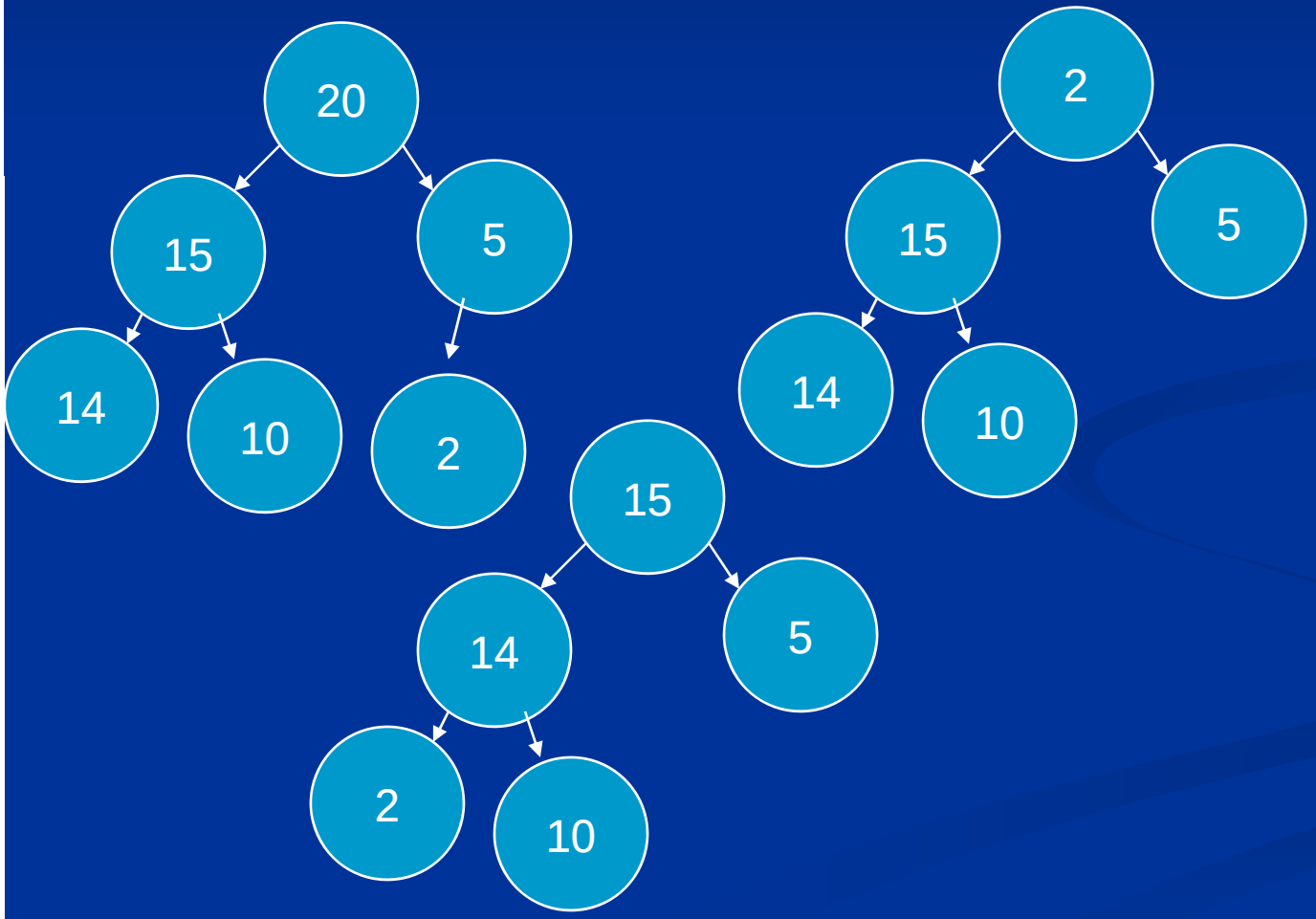
Max Heap: Insertion

```
template <class KeyType>
Void MaxHeap<KeyType>::Insert(const Element<Type> & toInsert)
{
    if (isFull()) return;
    currentSize = currentSize + 1;
    for (int i = currentSize; true; )
    {
        if (i == 1) break;          // if at root stop
        if (toInsert.key < heap[i/2].key) break;

        // if smaller than parent stop (preserve max tree property)

        heap[i] = heap[i/2];   // swap with parent
        i = i /2;                  // set parent position as potential insert
                                   // location
    }
    heap[i] = toInsert;         // insert at appropriate place
}
```

Max Heap: Insertion Big Oh Analysis

- الحاق :
  - نقطه شروع: یکی از برگها
  - نقطه انتها: حداکثر ریشه، ولی معمولا قبل از ریشه متوقف می شویم
  - حداکثر فاصله بین ریشه و برگها برابر $\log_2(n+1)$ است.
  - پیچیدگی هر جابجایی $O(1)$ است. (یک مقایسه دارد)
  - لذا، پیچیدگی زمانی الحاق **$O(\log n)$ است.**

Max Heap: Deletion



قانون اول:
ریشه را حذف کنید. (چون ریشه بزرگترین نود است)

قانون دوم:
خاصیت کامل بودن را رعایت کنید. پایین ترین و سمت راست ترین نود را در محل ریشه قرار بده.

قانون سوم:
خاصیت حداکثر بودن درخت را رعایت کنید. یعنی، از ریشه شروع کن و پدر را با فرزند بزرگتر جابجا کن.

Max Heap: Deletion Implementation

```
template <class KeyType>
Element<KeyType>* MaxHeap<KeyType>::DeleteMax(Element <KeyType> & x)
{
    if (isEmpty()) return 0;
    x = heap[1]; // grab root to return
    Element<KeyType> k = heap[currentSize];  // grab lowest rightmost item
    currentSize = currentSize – 1;
    for (int i= 1, j = 2; j <= currentSize; )
    {
        if (j < currentSize) if (heap[j].key < heap[j+1].key) j++;
        // j points to the larger child now
        if (k.key >= heap[j].key) break;    // in right place already
        heap[i] = heap[j];                  // move child up if not in right place
        i = j; j *= 2;                      // move i,j down to continue swapping
    }
    heap[i] = k;                            // put in right place
    return &x;
}
```

Max Heap: Deletion Big O Analysis

- حذف:
  - حذف ریشه: O(1)
  - جابجایی پایین ترین و سمت راست ترین نود با ریشه: O(1)
  - جابجایی پدر با فرزندان:
    - حداکثر به اندازه ارتفاع درخت $\log_2(n+1)$ پایین می رویم.
    - هر مرحله O(1) است (یک مقایسه).
- لذا، پیچیدگی زمانی حذف **O(log n) است.**

Priority Queue Big Oh Comparisons:

- پیاده کردن صف اولویت دار با heap:
  - الحاق: O(log n)
  - حذف: O(log n)
- پیاده کردن صف اولویت دار با آرایه نامرتب:
  - الحاق: O(1)
  - حذف: O(n)
- پیاده کردن صف اولویت دار با آرایه مرتب:
  - الحاق: O(n)
  - حذف: O(1)