درختان ۳

Heaps for searching

- جستجو در Heap:
  - به ریشه نگاه می کنیم.
  - اگر کوچکتر بود، به فرزندان سمت راست و چپ نگاه می کنیم.
  - از هر نودی که کوچکتر باشد، به فرزندان سمت راست و چپ آن نود نگاه می کنیم.
- ممکن است که تمام نودها را ببینیم، لذا، در بدترین حالت، جستجو $O(n)$ خواهد بود.
- زمان کل ساخت و استفاده از heap:
  - درست کردن heap: $nO(\log_2 n)$
  - جستجو: هر کدام $O(n)$ n
    - $nO(n)$
  - $O(n \log_2 n + n^2)$

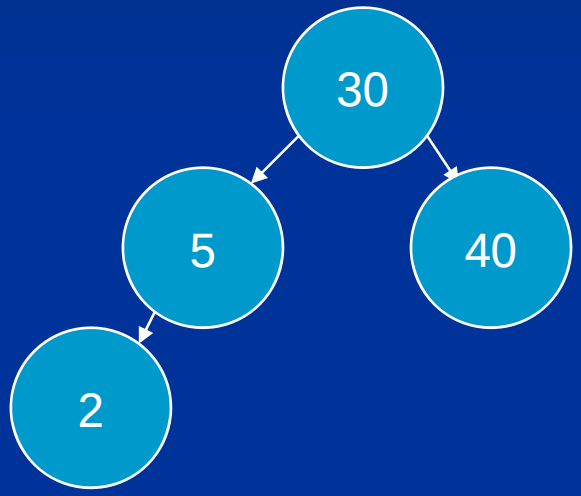
Heaps for Searching

- Heap فقط برای مسائلی خوب است که یک عنصر مشخص (مثل بزرگترین یا کوچکترین) مورد نظر باشد.
- می خواهیم راه حلی پیدا کنیم که جستجوی آن برای هر آیتم $O(\log n)$ باشد.
- البته در heap می توانیم تمام عناصر را تک تک حذف کنیم تا به یک لیست مرتب نزولی برسیم. به این روش Heapsort گفته می شود.
  - ساخت heap: $O(n\log_2 n)$
  - استخراج لیست مرتب: $O(n\log_2 n)$

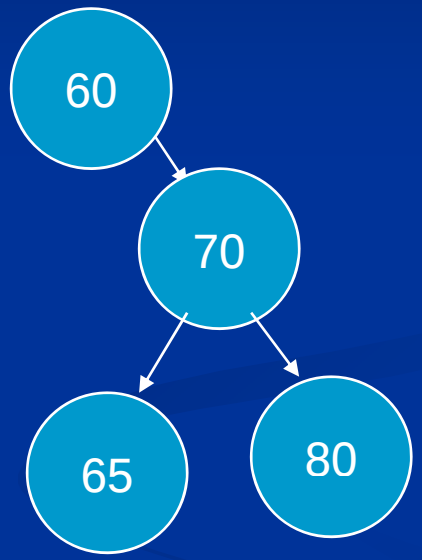
A Better Way to Search: Binary Search Trees

- درخت جستجوی دودویی:
  - درخت دودویی
  - صفر نود یا بیشتر
  - اگر 0< نود:
    - هر نود دارای یک کلید یکتا است.
    - کلید تمام نودهای زیر درخت سمت چپ نود، ازخود نود کمتر است.
    - کلید تمام نودهای زیر درخت سمت راست نود، ازخود نود بیشتر است.
    - زیر درختهای سمت چپ و راست نیز درخت جستجوی دودویی هستند.

Binary Search Trees

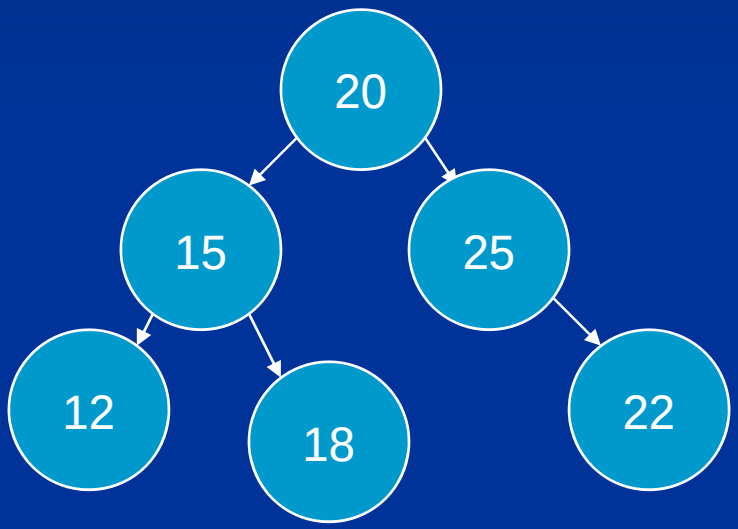


- کلیدهای یکتا
- نودهای سمت چپ کمتر از ریشه
- نودهای سمت راست بیشتر از ریشه
- زیر درختهای سمت چپ و راست نیز درخت جستجوی دودویی هستند.



- کلیدهای یکتا
- نودهای سمت چپ کمتر از ریشه
- نودهای سمت راست بیشتر از ریشه
- زیر درختهای سمت چپ و راست نیز درخت جستجوی دودویی هستند.

Binary Search Trees



درخت جستجوی دودویی نیست.
فرزند سمت راست ۲۵ ازخودش کمتر است.

- کلیدهای یکتا
- نودهای سمت چپ کمتر از ریشه
- نودهای سمت راست بیشتر از ریشه
- زیر درختهای سمت چپ و راست

درخت جستجوی دودویی نیستند.

Binary Search Trees

- دقت کنید که شرط کامل بودن در تعریف درخت جستجوی دودویی حضور ندارد.
  - لذا، پیاده سازی لینک پیوندی بهتر است.
- تعریف بازگشتی درخت جستجوی دودویی = الگوریتمهای بازگشتی

Binary Search Trees: Search

- جستجو:
  - از خواص درخت جستجوی دودویی استفاده کنید.

  - از ریشه شروع کن
  - اگر ریشه برابر صفر بود، پیغام بده که درخت خالی است.
  - در غیر این صورت:
    - x را با ریشه مقایسه کن.
    - اگر x با کلید ریشه برابر بود، نود را برگردان.
    - اگر x از کلید ریشه کمتر بود، زیر درخت سمت چپ را بگرد.
    - در غیر این صورت زیر درخت سمت راست را بگرد.

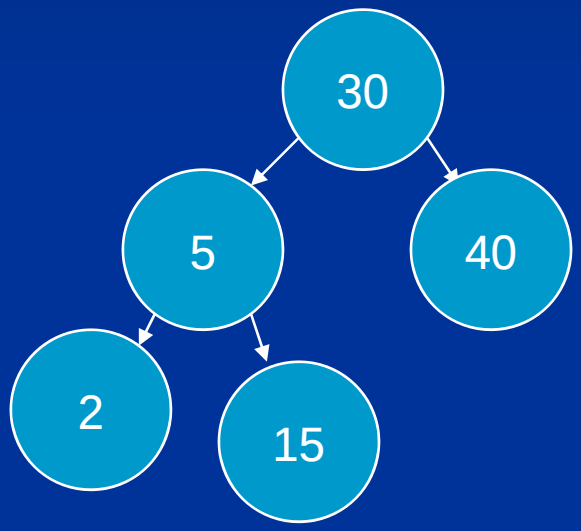
Binary Search Trees: BSTNode Definition

```
template <class Type>
class BSTNode
{
    private:
        BSTNode* leftChild;
        BSTNode* rightChild;
        Element<Type> data;
};

template <class Type>
class Element
{
    private:
        Type key;
        ??? OTHER DATA
}
```

Binary Search Tree: Search Implementation

```
template <class Type>
BSTNode<Type>* BST<Type>::Search(const Element<Type>& x)
{ return Search(root,x); }

template<class Type>
BSTNode<Type>* BST<Type>::Search(BSTNode*<Type> *b, const
    Element<Type>& x)
{
    if (b == 0) return 0;
    if (x.key == b->data.key) return b;
    if (x.key < b->data.key) return Search(b->LeftChild, x);
    return Search(b->rightChild, x);
}
```

Binary Search Trees: Search Example



۱۵ را پیدا کن.

ریشه خالی است؟ نه

۱۵ را با مقدار ریشه (۳۰) مقایسه کن

15 < 30 لذا زیر درخت سمت چپ را بگرد.

۱۵ را با ۵ مقایسه کن

15 > 5 لذا زیر درخت سمت راست را بگرد.

۱۵ را با ۱۵ مقایسه کن

15 == 15 نود جاری را بر گردان

Binary Search Trees: Big Oh Analysis

- در ریشه یک مقایسه انجام می دهیم
  - < Root یا > Root
- با توجه به نتیجه:
  - به یکی از فرزندان می رویم
  - یک مقایسه انجام می دهیم.

- حداکثر به اندازه ارتفاع درخت این کار را انجام می دهیم:
- لذا، پیچیدگی زمانی جستجو، وابسته به شکل درخت است.
  - خطی: $O(n)$
  - متوازن: $O(\log_2 n)$

Binary Search Trees: Insertion

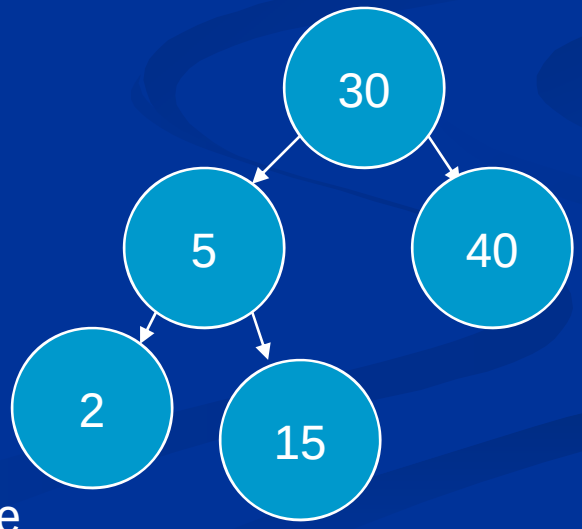
- قوانین – الحاق باید شرایط زیر را برآورده کند:
  - کلید یکتا
  - فرزند سمت راست > پدر
  - فرزند سمت چپ < پدر
  - نودهای میانی نیز باید شرایط فوق را برآورده کنند.
- یکتا بودن را چگونه چک کنیم؟
  - به همه نودها نگاه کنیم؟

Binary Search Trees: Insertion

- نیازی نیست که به تمام نودها نگاه کنیم
  - از این حقیقت استفاده می کنیم که قبل از الحاق نود جدید، درخت از نوع جستجوی دودویی است.
  - لذا کافیست دنبال نود جدید در درخت بگردیم.

Add

Search for 15
15 ? 30, 15 < 30 => Left
15 ? 5, 15 > 5 => Right
15 ? 15, 15 == 15 => Not Unique

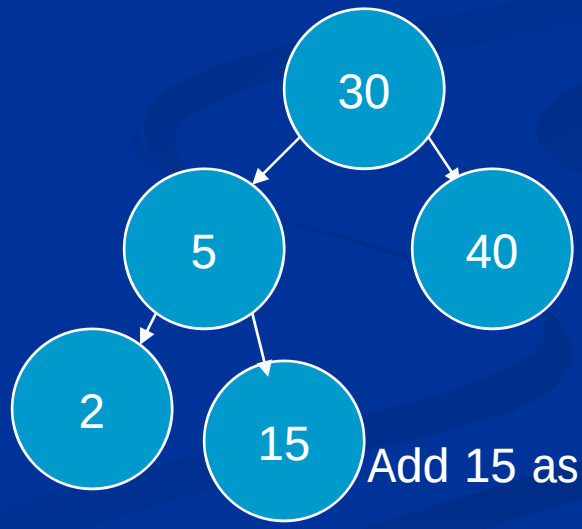


Binary Search Trees: Insertion

- جستجوی نود جدید نه تنها مساله یکتا بودن را حل می کند، بلکه ما را به جای درست نود جدید رهنمون می سازد.

Add

Search for 15
15 ? 30, 15 < 30 => Left
15 ? 5, 15 > 5 => Right
No right child, so not present



Add 15 as right child of 5

Binary Search Trees: Insertion Implementation

```
template <class Type>
bool BST<Type>::Insert(const Element<Type> & x)
{
    // search for x
    BSTNode<Type> *current = root; BSTNode<Type>* parent = 0;

    while (current) {
          parent = current;
          if (x.key == current-> data.key) return false;
          if (x.key < current->data.key) current = current->leftChild;
          else current = current->rightChild; }
    current = new BSTNode<Type>;
    current->leftChild = 0; current->rightChild = 0; current->data = x;
    if (!root) root = current;
    else if (x.key < parent->data.key) parent->leftChild = current;
    else parent->rightChild = current;
    return true;
}
```

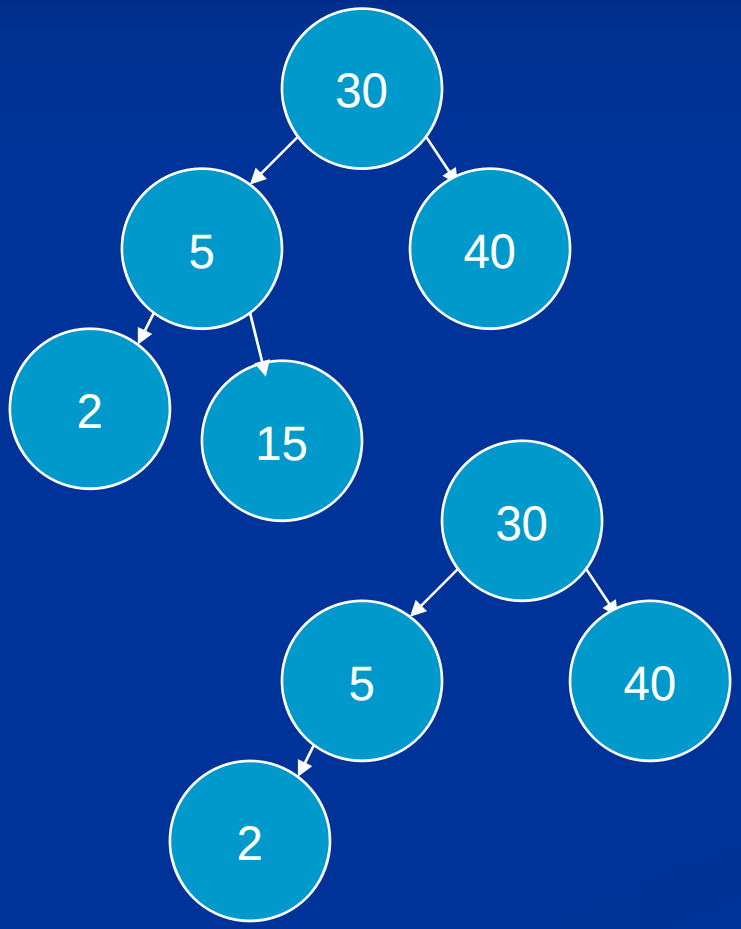
Binary Search Trees: Insertion Big Oh Analysis

- عمده کار تابع الحاق، پیاده سازی عمل جستجو است.
  - وابسته به شکل درخت است.
- خود عمل الحاق دارای هزینه ثابت است.
- لذا، هزینه کل وابسته به پیچیدگی عمل جستجو است.
  - در بدترین حالت: $O(n)$
  - در حالت میانگین: $O(\log_2 n)$

Binary Search Trees: Deletion

- قوانین – حذف باید شرایط زیر را برآورده کند:
  - کلید یکتا
    - نیازی به چک کردن ندارد. چون قبل از عمل حذف کلیدها یکتا هستند.
  - اما موارد زیر باید رعایت شوند:
    - فرزند سمت راست < پدر
    - فرزند سمت چپ > پدر
    - نودهای میانی نیز باید شرایط فوق را برآورده کنند.

Binary Search Trees: Deletion

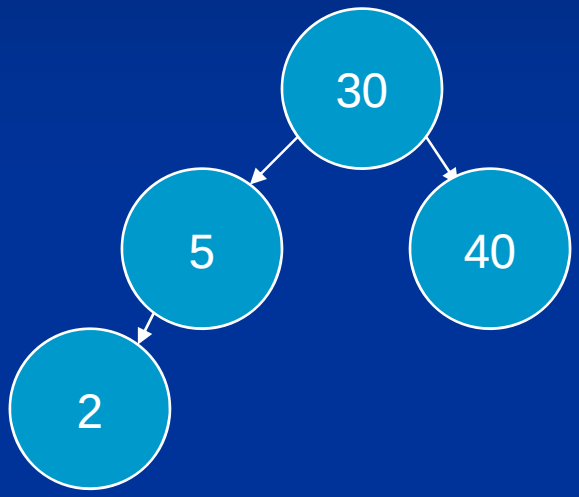


سه حالت:

۱- حذف نود انتهایی (15) :

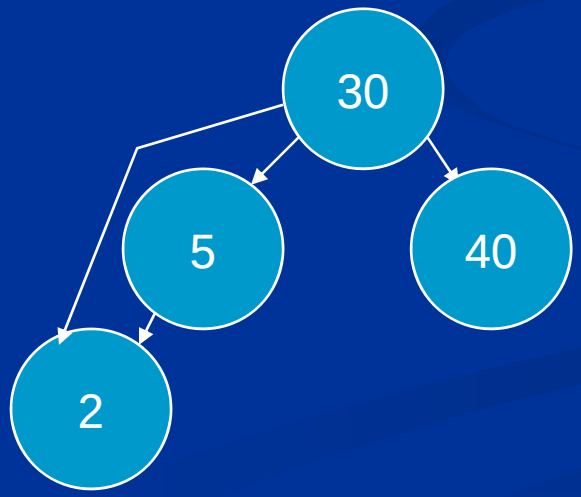
نود انتهایی را حذف کن.
اشاره گر پدر را برابر صفر قرار بده.

Binary Search Trees: Deletion



۲- نود غیر انتهایی، دارای یک فرزند(5):

لینک نود پدر را که به نود حذف شونده اشاره دارد را به فرزند نود حذف شونده اشاره دهید.



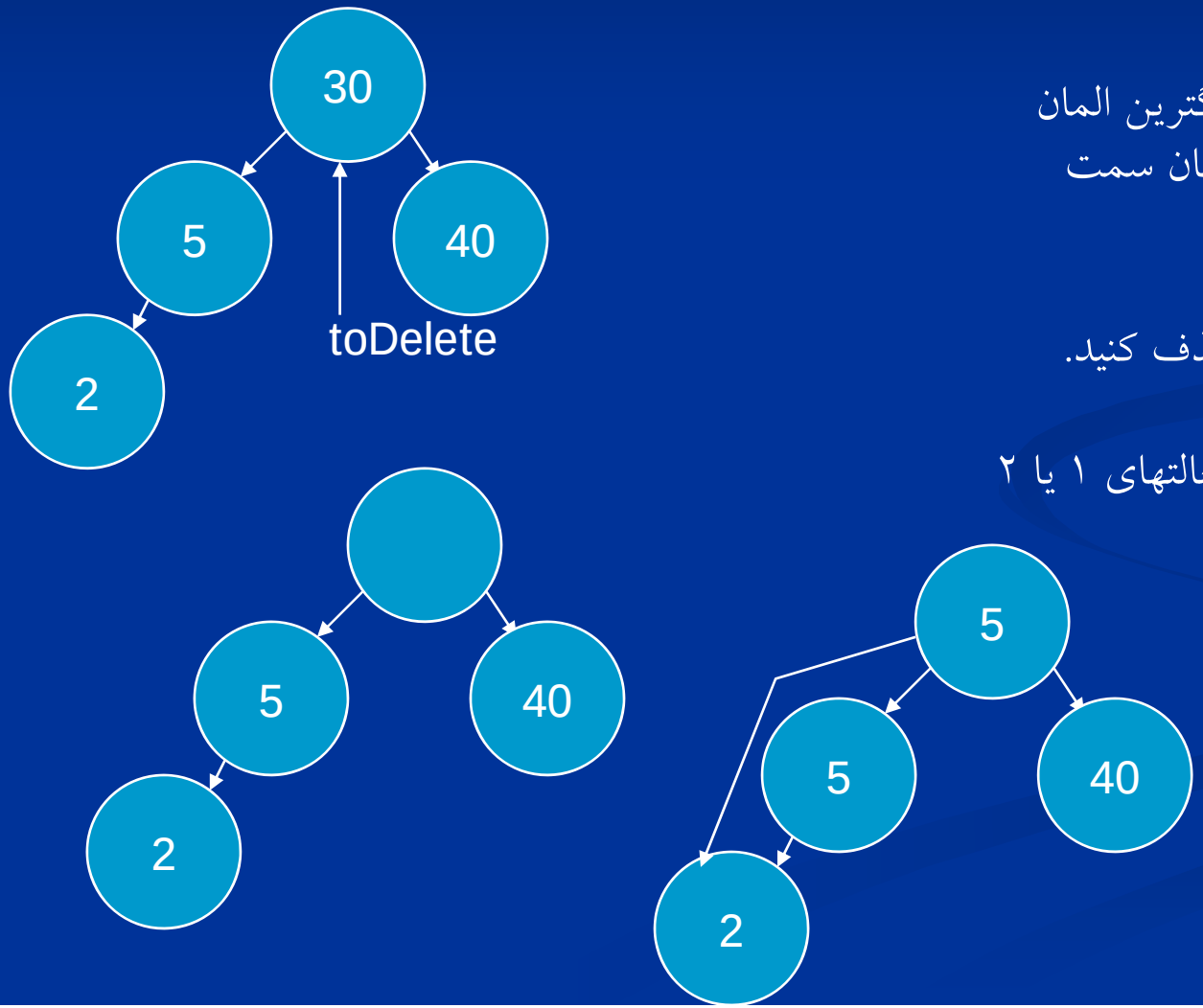
Binary Search Trees: Deletion

۳- نود غیر انتهایی دارای دو فرزند:

مقدار نود حذف شونده را با بزرگترین المان سمت چپ یا کوچکترین المان سمت راست جابجا کنید.

نودی را که جابجا کرده اید را حذف کنید.

این حذف کردن معادل یکی از حالتهای ۱ یا ۲ خواهد بود.



Binary Search Trees: Deletion

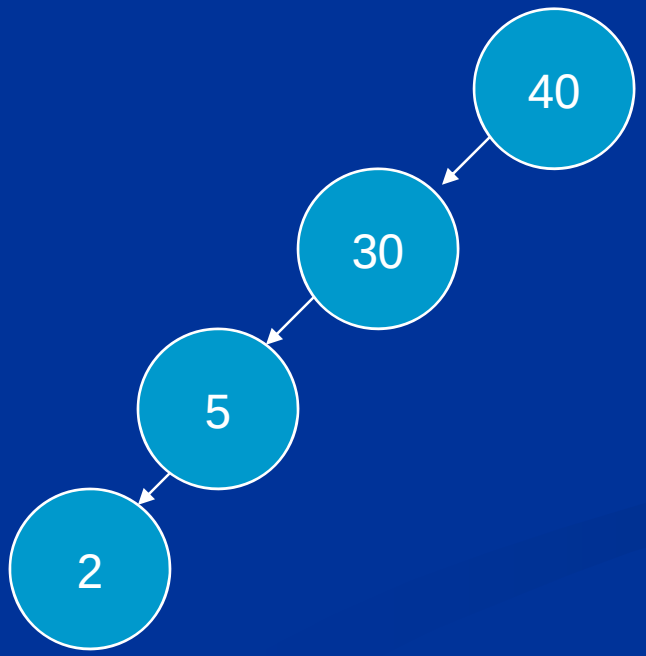
- قانون اصلی حذف:

"با بزرگترین المان سمت چپ یا کوچکترین المان سمت راست جابجا کنید"

- آیا این روش همیشه درست است؟
  - بله – چون بزرگترین المان سمت چپ
    - از تمام عناصر سمت چپ بزرگتر است.
    - از تمام عناصر سمت راست کوچکتراست.
  - کوچکترین المان سمت راست
    - از تمام عناصر سمت چپ بزرگتر است.
    - از تمام عناصر سمت راست کوچکتراست.
  - اینها، همان شرایطی هستند که ریشه زیر درخت جستجوی دودویی نیاز دارد.

Binary Search Trees: Height

- در بدترین حالت ارتفاع درخت باینری برابر n است.
  - درخت خطی



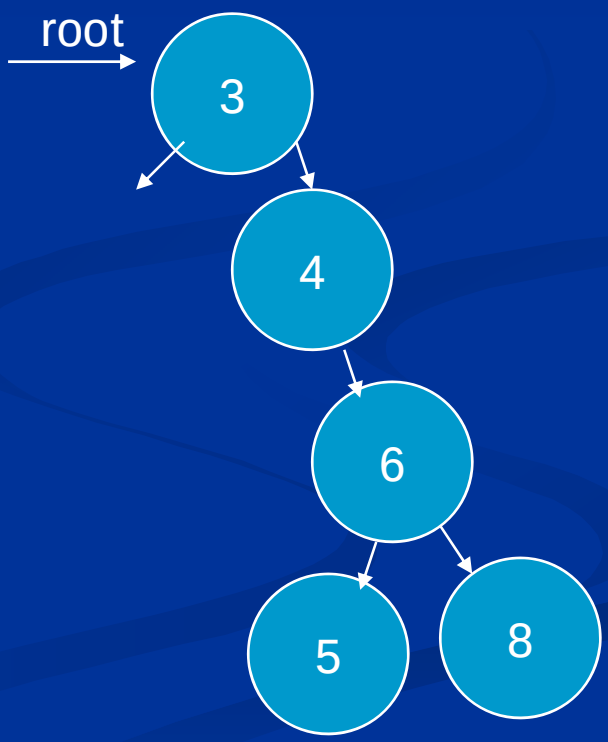
مسائل درخت جستجوی دودویی از لحاظ پیچیدگی وابسته به ارتفاع درخت هستند که در بدترین حالت O(n) هست.

اگر داده ها مرتب یا نیمه مرتب باشند، درخت خطی خواهد گردید.

Binary Search Trees: Height

```
bool BST<Type>::Insert(const Element<Type> & x)
{
    // search for x
    BSTNode<Type> *current = root; BSTNode<Type>*
    parent = 0;
    while (current) {
            parent = current;
            if (x.key == current-> data.key) return false;
            if (x.key < current->data.key) current = current-
    >leftChild;
            else current = current->rightChild; }
    current = new BSTNode<Type>;
    current->leftChild = 0; current->rightChild = 0; current-
    >data = x;
    if (!root) root = current;
    else if (x.key < parent->data.key) parent->leftChild =
    current;
    else parent->rightChild = current;
    return true;
}
```

Insert: 3, 4, 6, 5, 8

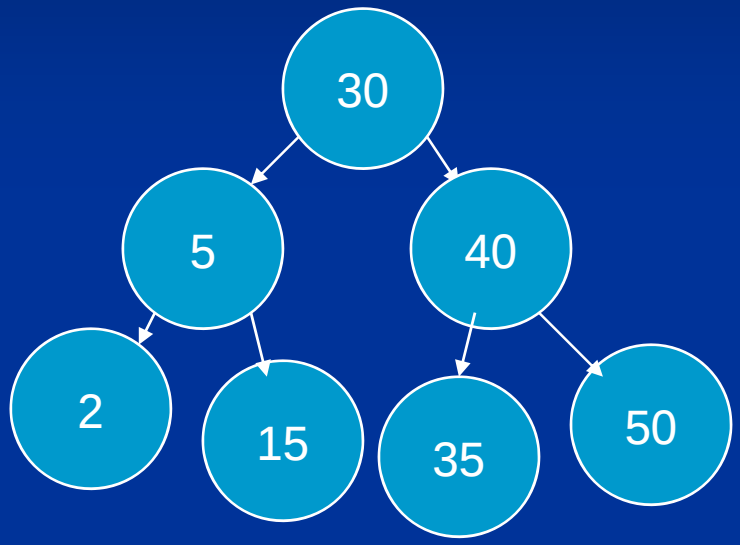


Binary Search Trees: Height

- اگر الحاقها به صورت تصادفی انجام گردند، ارتفاع درخت برابر $O(\log n)$ خواهد بود.
- در حالت عمومی الحاقها تصادفی هستند، لذا اغلب ارتفاع برابر $O(\log n)$ خواهد شد.
- راههای وجود دارد که ارتفاع $O(\log n)$ را گارانتی نمود. باید توابع الحاق و حذف را دستکاری نمود تا درخت متعادل شود.

TreeSort:

- نودهای درخت جستجوی دودویی دارای نظم خاصی هستند.
- تمام نودهای سمت چپ از ریشه کوچکتر و تمام نودهای سمت راست از آن بزرگتر هستند.
- این مطلب برای تمام نودها صادق است.
- لذا، می توان از پیمایش LVR برای تولید یک لیست مرتب استفاده کرد.



LVR Ordering:
2,5,15,30,35,40,50

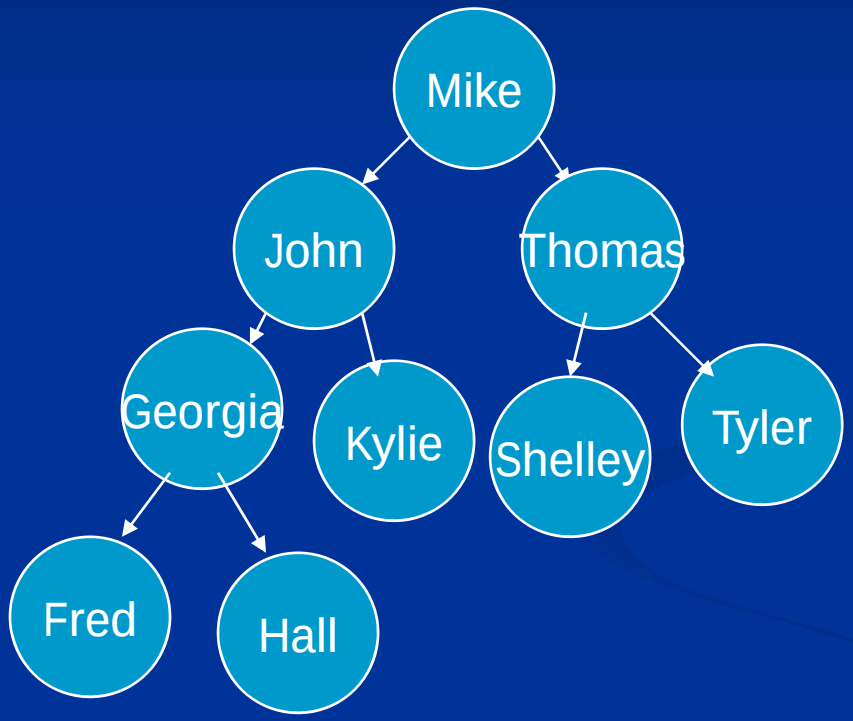
TreeSort:

- آنالیز TreeSort:
  - باید یک درخت جستجوی دودویی با سایز n بسازیم.
    - به n الحاق نیاز داریم.
      - بهترین حالت: درخت متعادل $O(n * \log_2 n)$
      - بدترین حالت: درخت خطی $O(n^2)$
  - سپس LVR را اجرا می کنیم.
    - همیشه $O(n)$ است.
  - پس پیچیدگی TreeSort برابر است با:
    - بهترین حالت: $O(n * \log_2 n)$
    - بدترین حالت: $O(n^2)$

TreeSort:

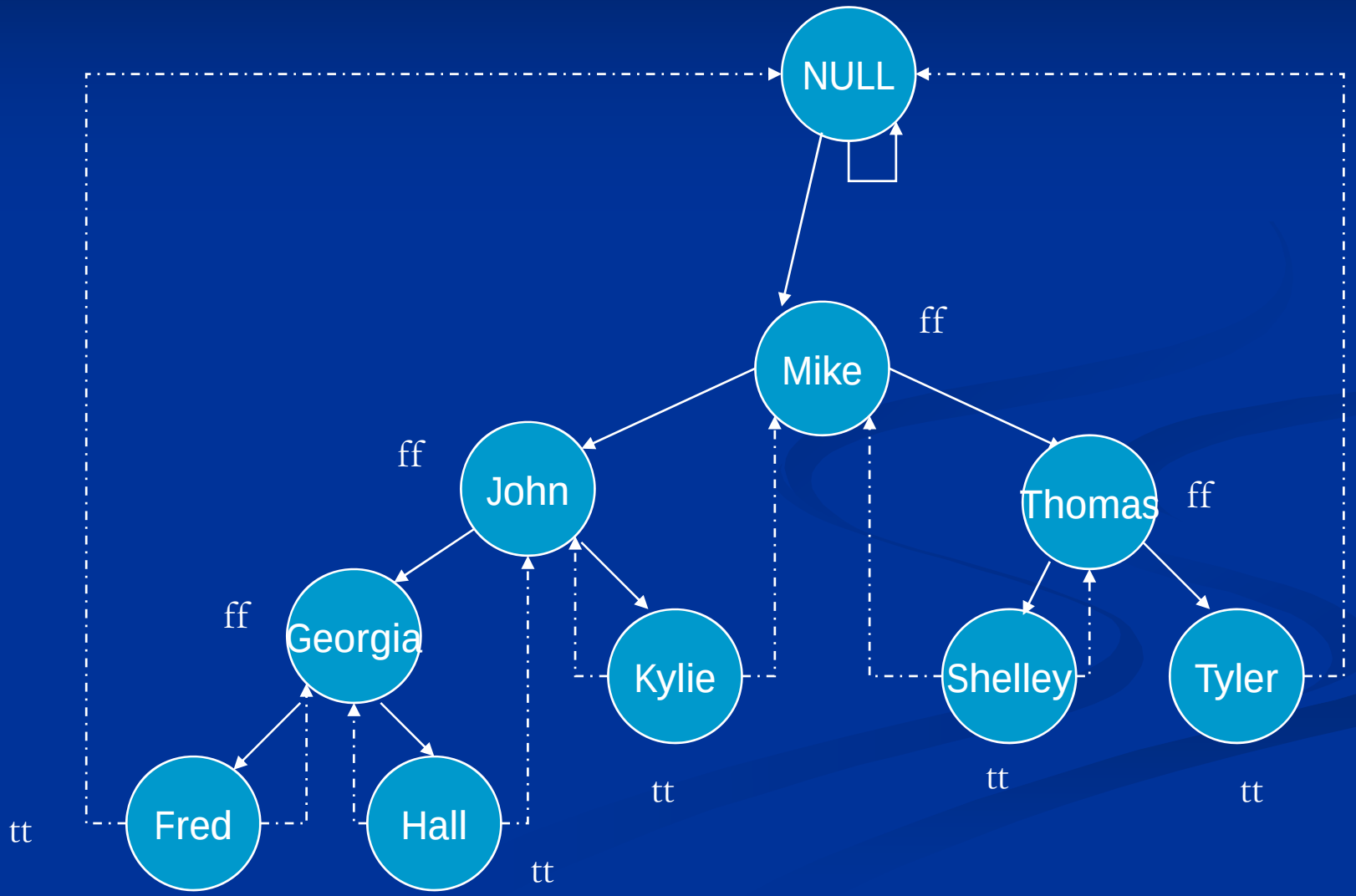
- خیلی شبیه quicksort است.
  - حالت میانگین $[O(n \log n)]$ و بدترین حالت $[O(n^2)]$ است.
  - ریشه هر زیر درخت معادل محور است.
  - عناصر کوچکتر از محور سمت چپ آن هستند.
  - عناصر بزرگتر از محور سمت راست آن هستند.
  - هر چقدر محور بهتر باشد، درخت متعادل تر است.
  - داده های مرتب یا نیمه مرتب هر دو روش را با مشکل مواجه می کند.
    - در quicksort: تفکیک با مشکل مواجه می شود.
    - در treesort: ساخت درخت با مشکل مواجه می شود.

Threaded Trees: General Trees

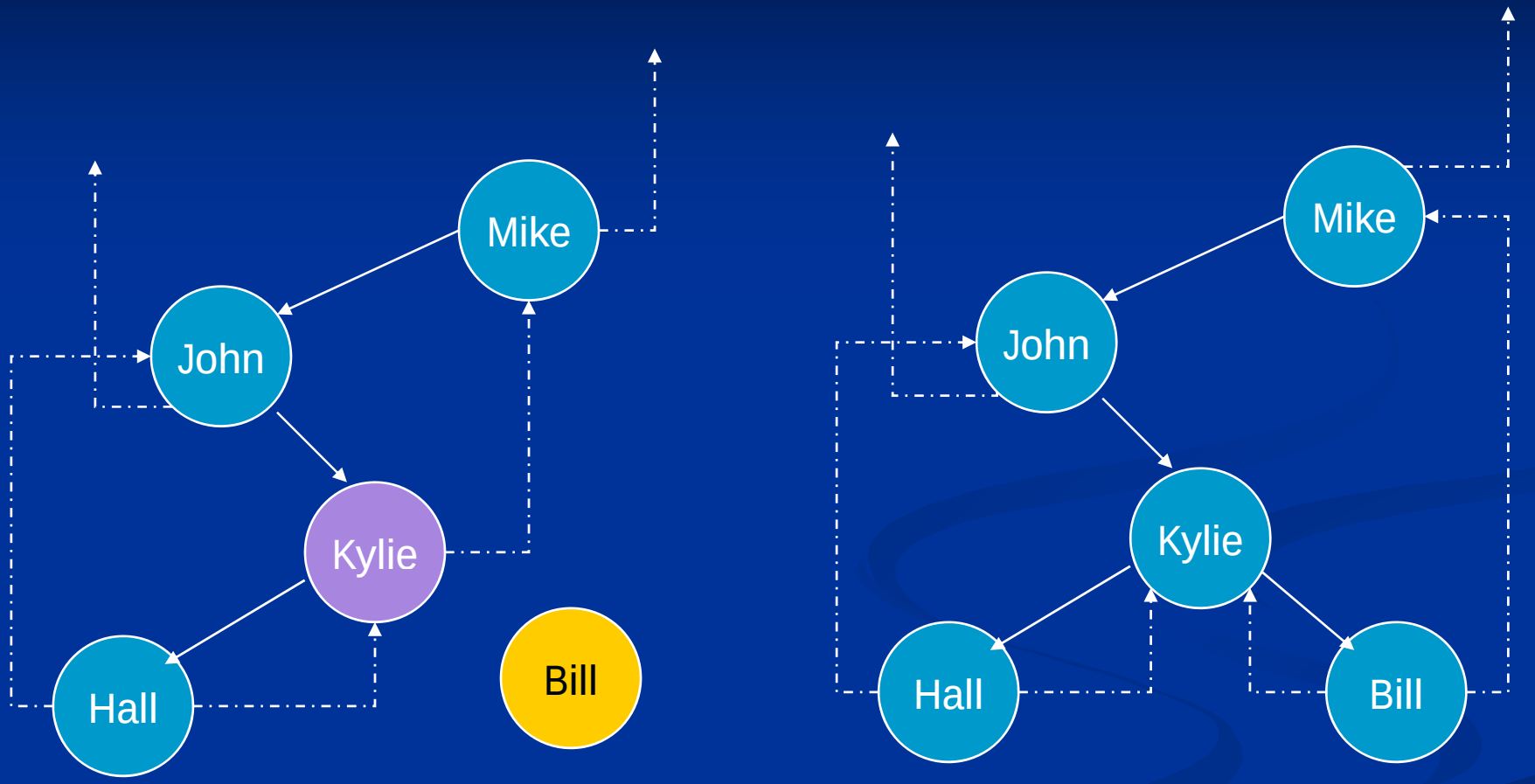


تعداد زیادی از لینکها برابر null هستند. می شود از این لینکها برای افزایش کیفیت پیمایش استفاده کرد.

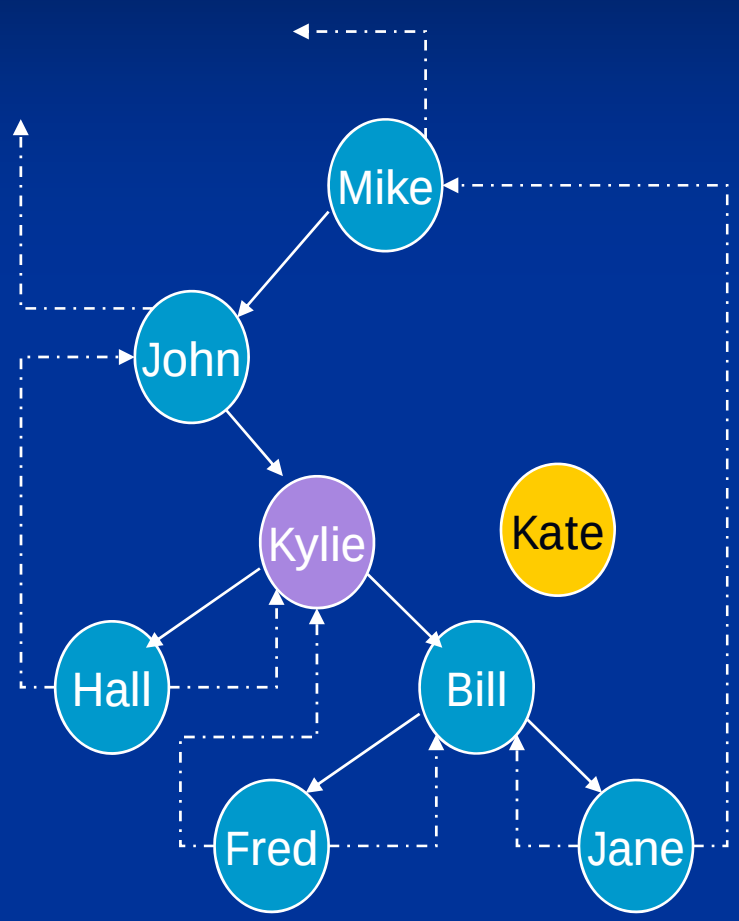
Threaded Trees

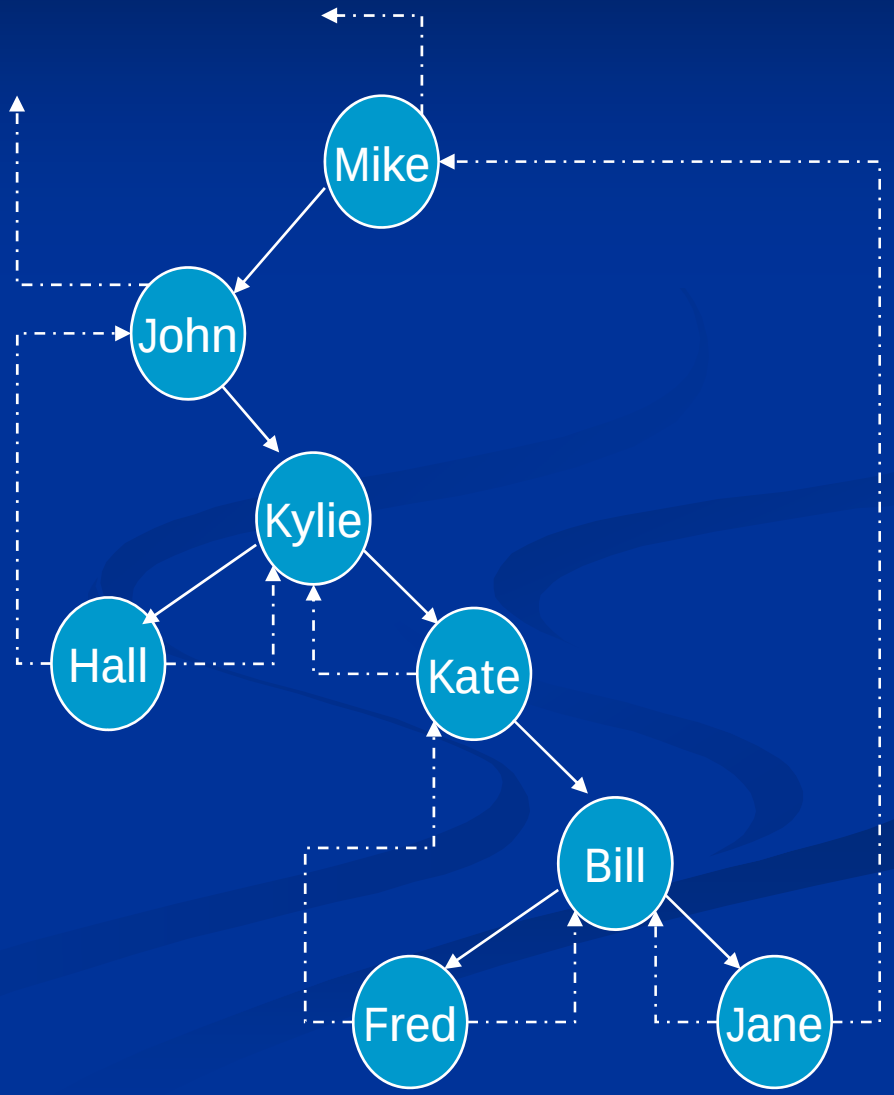


Threaded Trees: Insertion



Threaded Trees: Insertion





Rank Information

- اغلب اوقات ما به رتبه نود در لیست علاق داریم:
  - بزرگترین نود کدام است؟
  - کوچکترین نود کدام است؟
  - میانه کدام است؟
  - پنجمین نود از نظر بزرگی کدام است؟
- بزرگترین وکوچکترین معمولا با O(n) انجام می شوند. مگر اینکه از heap استفاده کنیم.
- اگر دنبال نودی غیر از بزرگترین (یا کوچکترین بودیم)، باید راهی موثر برای پیدا کردن رتبه داشته باشیم.

Rank Information

- راه حل اول پیدا کردن مرتبه:
  - مرتب کردن داده ها (لیست آرایه ای)
    - لیست را مرتب کنید. O(n log n)
    - مقدار list[rankOfInterest] را برگردانید. O(1)
    - O(n log n) [sort] + O(1) [value retrieval]
  - البته اگر داده دینامیک باشد از لیست پیوندی استفاده می شود.

Rank Information

- راه حل اول پیدا کردن مرتبه:
  - مرتب کردن داده ها (لیست آرایه ای)
    - لیست را مرتب کنید. (مثلا با mergesort)
  - لیست را تا رسیدن به نود مورد نظر پیمایش کنید.
- O(n log n) [sort] + O(rankOfInterest) [traversal]
- این روش با داده دینامیک کار می کند اما کندتر است.

Rank Information

- راه حل دوم پیدا کردن مرتبه:
  - استفاده از درخت جستجوی دودویی
    - داده ها را وارد درخت جستجوی دودویی کنید.
    - پیمایش میان ترتیب را تا رسیدن به مرتبه مورد نظر انجام دهید.
- O(n log n) [building tree] + O(rankOfInterest) [traversal]

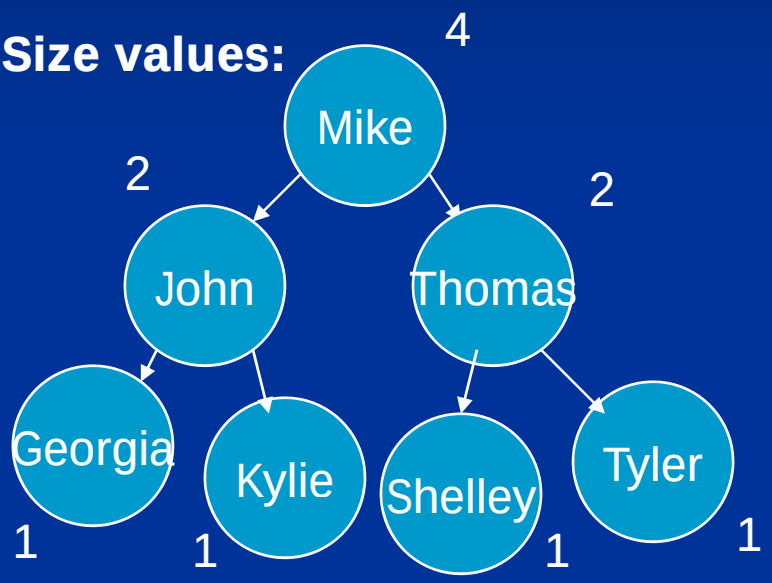
هزینه این روش مثل روش قبلی است اما دیگر نیازی به اجرای mergesort برای لیست پیوندی نیست.

Rank Information:

- راه حل سوم پیدا کردن مرتبه:
  - استفاده از درخت جستجوی دودویی
  - یک متغییر به هر نود اضافه کنید.
    - این متغییر نشان دهنده تعداد نودهای سمت چپ نود به اضافه یک (به خاطر خود نود) است.
    - ابتدا مقدار این متغییر یک است.
  - وقتی که یک نود به درخت اضافه می شود:
    - از هر نود که می گذریم، یک اشاره گر به پدر آن نود نگهداری میکنیم.
    - اگر نود را به درخت اضافه کردیم. مقدار متغییر را برای والدین یک واحد افزایش می دهیم.
  - نحوه استفاده از این متغییر در صفحه بعد نشان داده شده است.

Rank Information: Example

leftSize values:



Real Ranks for Data [First is rank 1, Last is 7]:
Georgia, John, Kylie,
Mike, Shelley, Thomas, Tyler

دومین عنصر کیست؟

Rank 2 < leftSize(Mike) [4]
Move to root->leftChild
Rank 2 == leftSize(John) [2]

Return John Node

پنجمین نود کیست؟

Rank 5 > leftSize(Mike) [4]
Move to root->rightChild
Rank = 5-4 = 1 < leftSize(Thomas) [2]
Move to leftChild of Thomas
Rank == leftSize(Shelley) [1]

Return Shelley Node

Rank Information:

```
template <class Type>
BinaryTreeNode<Type>* BinarySearchTree<Type>:: search(int
    rank)
{
    BinaryTreeNode<Type>* current = root;
    while (current)
    {
          if (rank == current->leftSize) return current;
          else if (rank < current->leftSize) current =
                                                current->leftChild;
          else  { rank = rank – current->leftSize; current =
                                                current->rightChild;}
    }
}
```

Rank Information: Analysis

- حالا محدود به ارتفاع درخت هستیم.
  - به طور متوسط O(log n)
- ساخت درخت قبلا O(n log n) بود.
- اما ما به درخت یک متغییر و تعدادی پردازش اضافه کردیم:
  - تغییر دادن مقدار leftsize والدین
    - حداکثر به اندازه ارتفاع درخت.
  - پس هر الحاق به 2*logn کار نیاز دارد. لذا n الحاق به O(n log nکار نیاز دارد.)
- پس برای داده دینامیک و مساله پیدا کردن رتبه داریم:
- O(n log n) [building] + O(log n) [searching]
- از روشهای قبلی بهتر است.