

EECS 560

DATA STRUCTURES

MODULE 0: THE FUNDAMENTALS

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++*, 4th edition, by Mark Allen Weiss.

ADT (ABSTRACT DATA TYPE)

- ADT is an mathematical abstraction of a data structure, containing the set of its objects (data) and operations (methods).
- Example:
 - “vector” is an ADT
 - Its object primarily contains an array
 - Its methods may include “retrieve(i)”, which returns the value of its i th element, and “size()”, which returns the number of elements in the vector

ADT CONT.

- The primary goal of this course is to gain a deep understanding of the implementational details of the major ADTs (both of the data and methods).
- The secondary goal is to analyze the efficiency for the methods of different ADTs.
- We will be working on:
 - Simple ADTs: list, queue, stack, and hash table
 - Tree-based ADTs: search trees, heaps, disjoint sets
 - Graph ADTs
 - Sorting algorithms

FUNDAMENTALS

- In this module we will be reviewing fundamentals for learning data structure
 - C++ features and general guidelines for implementing data structures
 - basic concepts in time complexity analysis

C++ FEATURES

- We will be discussing the following C++ features:
 - pointers and references
 - parameter passing
 - C++ objects
 - the big five: fundamental methods for data structure ADTs
 - template
 - function as parameter and function object

C++ FEATURES: POINTER

- Pointer is a variable that stores the (memory) address where another object resides.
- Assume we have an object named IntCell, we can use “*” to indicate that the variable is an object:

```
1 int main( )
2 {
3     IntCell *m;           // m is a pointer that points an address that stores the IntCell type
4
5     m = new IntCell{ 0 };
6     m->write( 5 );
7     cout << "Cell contents: " << m->read( ) << endl;
8
9     delete m;
10    return 0;
11 }
```

C++ FEATURES: POINTER

- Unlike many programming languages; C++ relies on the programmer to manage the memory usage for higher flexibility and efficiency (while being more challenging to program and debug)
 - memory allocation with `new`: e.g., `int *x = new int;`
 - memory collection with `delete`: e.g., `delete x;`
- Ideally, any address we have allocated must be reclaimed. However, in many cases this is not the case:
 - e.g., if we alter `x` by adding `x = nullptr` before reclaiming the address it points to, the program will lose the track of where `x` originally points to, and failed to reclaim the corresponding memory block
 - this is often called a **memory leak**, a common error found in many C++ programs

C++ FEATURES: POINTER

- Consider the example where `x` is a pointer for integer type:
 - You can access the data that `x` points to with “`*`”
 - for example: `std::cout << *x << std::endl;` will print out 4
 - note that `std::cout << x << std::endl;` will print out 0001
 - You can modify the data that `x` is pointing at with “`*`”
 - for example, `*x=2` will replace the 4 in address 0001 with 2
 - note that `x=2` will set `x` to point to address 0002
 - You can assign a pointer to another pointer
 - for example, `y=x` will assign `y` with a value of 0001; in this case both `y` and `x` are pointing to the address 0001
 - if we only want to assign the value `x` is pointer at to the address where `y` is pointer at, we should use `*y=*x`

	address
3	0000
4	0001
1	0002
8	0003
6	0004

C++ FEATURES: POINTER

- We can get the address of a variable using operator “&”, and assign it to a pointer:
 - e.g., `int a = 6; int *x = &a;`
 - now x will point to a
- If the pointer points to an object, we can access its member through operator “->”
 - e.g., `IntCell *x = new IntCell; x->findMax();`
 - x will be a pointer for the object IntCell, and we can use “->” to call IntCell’s member function `findMax()`

C++ FEATURES: REFERENCE

- References can be considered as alias of variables (lvalue) or constants (rvalue).
- Reference can be defined using the operator “&”
 - e.g., `int &x = a;`
 - in this case, we can use `x` interchangeably with `a`
- The major difference between reference and pointer:
 - reference **does not need a memory space to store**; it only takes up a small space in stack
 - pointer **does need a memory space to store**; it also takes up a small space in stack

C++ FEATURES: REFERENCE

- Some other important differences between pointer and reference:
 - pointer can be re-assigned, reference cannot
 - `int *p = nullptr; p = &a;` is valid; while `int &p = a; p = b;` is invalid
 - you can have multi-level indirection in pointer (such as pointer of pointer of ... a type), but you can have only one-level indirection for reference
 - `int *y = a; int **x = y;` is valid; while the operator “`&&`” would mean a different thing (rvalue reference, will discussed in detail later) when declaring reference
 - pointer can be declared uninitialized, reference cannot
 - `int *x;` is valid; while `int &x;` is invalid

C++ FEATURES: REFERENCE

- More differences between pointer and reference:
 - pointer can be modified and used to iterate an array; reference cannot
 - `int *p = a[0]; p++;` is valid; while `int &p = a[0]; p++;` is invalid
 - pointer needs to be dereferenced when accessing the memory location it points to; reference does not need to be dereferenced
 - `int *p = a; cout << *p;` vs `int &p = a; cout << p;`
 - pointer can be stuffed into arrays, reference cannot
 - `int **p = new int* [n];` is valid; while we cannot have an array of references
 - pointer cannot be bound to temporaries (rvalue), reference can
 - `int *p = 12;` is invalid; while `int &p = 12;` is valid

C++ FEATURES: REFERENCE

- Reference can be further divided into reference to **lvalue** and reference to **rvalue**:
 - lvalue: regular variables, which are stored in some memory block; for example, `int a, int *p, r[4]` etc.
 - rvalue: temporary constants, which are not stored in memory block: for example, `12, "hello world"` etc.
 - (since pointer needs to point to an address and rvalues do not have addresses, so there is not pointer to rvalue)
 - C++ allow reference to both lvalue (with “&”) and rvalue (with “&&”)
 - lvalue reference: `int &p = a;`
 - rvalue reference: `int &&p = 12;`

C++ FEATURES: REFERENCE

- How C++ handles lvalue reference and rvalue reference:
 - for lvalue reference, C++ will simply create an alias of the address as the reference (note that lvalues are already stored in memory); when using, we must first **copy** the constant value to a new address (and we have two copies of the data, one copy as temporary data, and another copy as stored in memory), and the reference can be created for the copy stored in memory.
 - for rvalue reference, C++ will first **move** the temporary value in memory and create a corresponding alias for it; (we only have one copy of the data) as a result this mechanism is faster when passing constant parameters (will be discussed in later slides).

C++ FEATURES: PARAMETER PASSING

- There are four major ways to pass parameters to C++ functions
 - call by value
 - call by constant reference
 - call by lvalue reference
 - call by rvalue reference

C++ FEATURES: PARAMETER PASSING

- Call by value:
 - needs to make a copy of the parameter before passing into the function (slow)
 - cannot be used to modify the parameters (non-mutable)

```
double average( double a, double b );    // returns average of a and b
void swap( double a, double b );        // swaps a and b; wrong parameter types
string randomItem( vector<string> arr ); // returns a random item in arr; inefficient
```

C++ FEATURES: PARAMETER PASSING

- Call by constant reference
 - can avoid copying the variables (fast)
 - keeps the variables unchanged (non-mutable)

```
string randomItem( const vector<string> & arr ); // returns a random item in arr
```

C++ FEATURES: PARAMETER PASSING

- Call by lvalue reference:
 - can avoid copying variables (fast)
 - allows modification of variables (mutable)
 - perhaps the mostly-used parameter passing mechanism

```
void swap( double & a, double & b );      // swaps a and b; correct parameter types
```

C++ FEATURES: PARAMETER PASSING

- Call by rvalue reference
 - allows passing of constants as parameters (which call by value can do but call by lvalue reference cannot)
 - can avoid copying the constants (which call by value cannot do but call by lvalue reference can)

```
string randomItem( const vector<string> & arr ); // returns random item in lvalue arr
string randomItem( vector<string> && arr ); // returns random item in rvalue arr

vector<string> v { "hello", "world" }; // we don't have to copy the strings to v if we use rvalue reference
cout << randomItem( v ) << endl; // invokes lvalue method
cout << randomItem( { "hello", "world" } ) << endl; // invokes rvalue method
```

C++ FEATURES: PARAMETER PASSING

- A note on call by passing pointer:
 - an older parameter passing style in the C age
 - in most cases, it has the same effect as call by lvalue reference
 - when to use call by passing pointer?
 - if you do need to take advantage of the features of pointer that reference cannot offer; e.g., looping through a vector, modify the pointer, allowing NULL pointer (uninitialized parameters) etc.
 - if you are working on legacy C code and wish to keep the coding style consistent

C++ FEATURES: OBJECT

- As an object-oriented programming language, the key components in C++ programs are objects.
- In this course, we will implement each data structure ADT as an object.

C++ FEATURES: OBJECT

- Some coding practice:
 - Each object can contains **data** and **methods** (while does not need to contain both). For example, for a vector object, we can use an array as data the store the items of the vector, and implement a method that finds the maximum item from the vector. Data and methods are also called **members** of the object.
 - Both data and methods can be declared as either **private** or **public**. Private members are not directly accessible by methods that are not a member of the object, while public members are.
 - A C++ object is often split into two parts: interface and implementation. Interface defines the **signature** (data type, parameter types, and return type) of the components and is often written in a **.h** file (also called header file), while implementation actually implement the methods and is often written in a **.c**, **.cc**, or **.cpp** file.

C++ FEATURES: THE BIG FIVE

- For most of C++ object implementations (and every data structure ADT we will be implementing in this course), they all contain five methods that will provide a comprehensive and standard interface. These five methods are called “the big five”:
 - Destructor: reclaim allocated memory to the object
 - Copy constructor: initialize the object from another object
 - Move constructor: initialize the object from rvalue
 - Copy assignment: overwrite the object using another object
 - Move assignment: overwrite the object using rvalue
 - The major difference between copy and move methods: copy methods will keep the original variable, while move will not. As a result, move methods are intended to be used with rvalue parameters.

C++ FEATURES: THE BIG FIVE

- Initialization list in C++ 11:
 - a colon followed by a list of comma-separated member initializers
 - the only way to initialize a `const` member in the object
 - the only way to initialize a member object if the member object has no zero-parameter constructor

```
class X {  
    int a, b, i, j;  
public:  
    const int& r;  
    X(int i)  
        : r(a) // initializes X::r to refer to X::a  
        , b{i} // initializes X::b to the value of the parameter i  
        , i(i) // initializes X::i to the value of the parameter i  
        , j(this->i) // initializes X::j to the value of X::i  
    {}  
};
```

C++ FEATURES: THE BIG FIVE

- You may notice that some member functions are initialized with braces “{}” while the other ones are initialized with parenthesis “()”.
- Using braces “{}” is a recent style encouraged by C++ to consolidate initialization syntax. In many cases using braces and parenthesis are the same.
- However, in some cases they are different.
 - they may invoke different object constructors, depending how they are defined.
 - initialization with parenthesis “()” is considered as calling the constructor with **N parameters**
 - initialization with braces “{}” is considered as calling the constructor with **one list parameter**

C++ FEATURES: THE BIG FIVE

```
1 int main() {  
2     std::vector<int> a(10, 5);  
3     std::vector<int> b{10, 5};  
4  
5     std::cout << a.size() << "-" << a[1] << std::endl;  
6     std::cout << b.size() << "-" << b[1] << std::endl;  
7 }
```

The output of this program is:

```
1 10-5  
2 2-5
```

This is because `std::vector` defines different constructors for accepting two integer parameters and accepting one list parameter. The effect for the first call would be initializing a vector with ten fives, while the second call would be initializing a vector with a ten and a five.

C++ FEATURES: THE BIG FIVE

```
IntContainer(int s, int v) {  
    size = s;  
    ints = new int[s];  
  
    for (int index = 0; index < size; ++index) {  
        ints[index] = v;  
    }  
}
```

constructor with two parameters

```
IntContainer(std::initializer_list<int> list) {  
    size = list.size();  
    ints = new int[size];  
  
    auto it = list.begin();  
    for (int index = 0; index < size; ++index, ++it) {  
        ints[index] = *it;  
    }  
}
```

constructor with one list parameter

C++ FEATURES: THE BIG FIVE

data declaration

```
34     private:  
35         int *storedValue;
```

the big five

```
~IntCell( ) // Destructor  
{ delete storedValue; }  
  
IntCell( const IntCell & rhs ) // Copy constructor  
{ storedValue = new int{ *rhs.storedValue }; }  
  
IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue } // Move constructor  
{ rhs.storedValue = nullptr; }  
  
IntCell & operator= ( const IntCell & rhs ) // Copy assignment  
{  
    if( this != &rhs )  
        *storedValue = *rhs.storedValue;  
    return *this;  
}  
  
IntCell & operator= ( IntCell && rhs ) // Move assignment  
{  
    std::swap( storedValue, rhs.storedValue );  
    return *this;  
}
```

A more efficient swap via taking advantages of rvalue

C++ FEATURES: THE BIG FIVE

the less efficient way

```
void swap( vector<string> & x, vector<string> & y )  
{  
    vector<string> tmp = x;  
    x = y;  
    y = tmp;  
}
```

three copies

the more efficient way

```
void swap( vector<string> & x, vector<string> & y )  
{  
    vector<string> tmp = std::move( x );  
    x = std::move( y );  
    y = std::move( tmp );  
}
```

three reference changes:

- `std::move()` marks the variable as rvalue, which means that it can be moved without copying.
- the first line assigns `tmp` to label the data `x`
- the second line assigns `x` to label the data `y`
- the third line assigns `y` to label the data `tmp`

C++ FEATURES: TEMPLATE

- Template allows for a single implementation of the same behaviors on different data types:
 - function template
 - class template
- For example:
 - find the maximum number of an array of integers
 - find the maximum number of an array of floats
 -
- For most of the data structure ADTs, we would like to implement them as templates to make them more generic.

C++ FEATURE: TEMPLATE

```
1  /**
2   * Return the maximum item in array a.
3   * Assumes a.size( ) > 0.
4   * Comparable objects must provide operator< and operator=
5   */
6 template <typename Comparable>
7 const Comparable & findMax( const vector<Comparable> & a )
8 {
9     int maxIndex = 0;
10
11    for( int i = 1; i < a.size( ); ++i )
12        if( a[ maxIndex ] < a[ i ] )
13            maxIndex = i;
14    return a[ maxIndex ];
15 }
```

A function template

C++ FEATURE: TEMPLATE

```
1  /**
2   * A class for simulating a memory cell.
3   */
4  template <typename Object>
5  class MemoryCell
6  {
7      public:
8          explicit MemoryCell( const Object & initialValue = Object{ } )
9              : storedValue{ initialValue } { }
10         const Object & read( ) const
11             { return storedValue; }
12         void write( const Object & x )
13             { storedValue = x; }
14     private:
15         Object storedValue;
16     };

```

A class template

C++ FEATURES: TEMPLATE

- Generic programming:
 - Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.
 - That is, the use of template
- Technically, the declaration and implementation of a template object must be put in the same file (.h). In other words, if you use template, then you will write all your code in the .h file and will not need the .cc file.
- Generic programming is more extensible, but also needs a bit more effort to maintain
- C++ STL adopted the generic programming style

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

- Occasionally we may want to use different ways to handle the same data, and don't want to re-implement the function:
 - imagine a function that finds the largest rectangles among a set of them
 - we can compare them based on their areas
 - we can also compare them based on their perimeters
- One solution would be to write two comparison functions, and pass the comparison function as parameter. For example, `findMax(rectangles, compareByArea)` or `findMax(rectangles, compareByPerimeter)`.
- A more recent style is to implement the functions as objects

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

- An example based on std::sort():

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
function definition
```

```
|bool myfunction (int i,int j) { return (i<j); }
comparison function definition
```

```
// using function as comp
std::sort (myvector.begin()+4, myvector.end(), myfunction);
passing the comparison function as parameter
```

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

```
// Generic findMax, with a function object, C++ style.  
// Precondition: a.size( ) > 0.  
template <typename Object, typename Comparator>  
const Object & findMax( const vector<Object> & arr, Comparator isLessThan )  
{  
    int maxIndex = 0;  
  
    for( int i = 1; i < arr.size( ); ++i )  
        if( isLessThan( arr[ maxIndex ], arr[ i ] ) )  
            maxIndex = i;  
  
    return arr[ maxIndex ];  
}
```

the `findMax()` function

```
class CaseInsensitiveCompare  
{  
public:  
    bool operator( )( const string & lhs, const string & rhs ) const  
        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }  
};
```

the comparison function object

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

```
int main( )
{
    vector<string> arr = { "ZEBRA", "alligator", "crocodile" };

    cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
    cout << findMax( arr ) << endl;

    return 0;
}
```

calling `findMax()` with function object as parameter

TIME COMPLEXITY ANALYSIS

- Time complexity analysis refers to the evaluation of the efficiency of the algorithm.
- Many complications exists:
 - primitive operations
 - speed of the machine
 - implementation (software and hardware)
 - input size (measured by the number of bits)
 - ...
- As a result, we will be developing a model to eliminate the impact of these confounding factors.

TIME COMPLEXITY ANALYSIS

- Time complexity model:
 - the model is a function that converts the input size to the number of primitive steps
 - only count the number of primitive operations (e.g., addition is much faster than multiplication, but we count either operation as one primitive operation), which leads to the elimination of coefficient in our study
 - disregarding term coefficient: consider two terms $5x$ and $10x$
 - the algorithm having the second term as time complexity can run faster if it is assigned to a faster machine
 - focus on the cases where the input size is large, which leads to only focusing on the fastest-growing term of the function disregarding the specific constant coefficient
 - fastest-growing term: consider two terms: $0.001x^2$ and $1000x$
 - we see that the first term will eventually dominate when x grows sufficiently large

TIME COMPLEXITY ANALYSIS

- We can consider time complexity, when multiplied by a large-enough constant, as an upper bound of the running time with arbitrarily large input size.
 - consider a function that converts the size of input, n , to the number of primitive operations, T :
 - $T = 16n^2 + 8n + 17$; note that n is an integer and $n > 0$
 - by eliminating the coefficient and only focusing on the fastest-growing term, we get n^2
 - if we multiply n^2 with a large-enough constant, say 41 ($=16+8+17$), we will get $41n^2$, which is guarantee to be larger than T for arbitrarily large n
 - can we pick a slower-growing term as the complexity? say n ? Note that no matter how large the constant we will multiply, it will fail to upper bound T with large n .
 - $999999n$ will fail to bound T if n is larger than 999999

TIME COMPLEXITY ANALYSIS

- The above upper bound is most-frequently used in time complexity analysis.
- It measures (in a pessimistic point of view) how slower the program will run when the input size grows.
- We use big-O to denote such an upper bound:
 - in the previous example where $T = 16n^2 + 8n + 17$, we will write $T = O(n^2)$
 - we say “the running time of the program/algorithm is upper bounded by the order of n^2 ”
 - or, in many cases, we simply say that “the program/algorithm runs in n^2 time”

TIME COMPLEXITY ANALYSIS

- In addition to upper bound, we also have lower bound and tight bound.
- Lower bound measures the minimum amount of time required by the program.
- Tight bound is established when the upper bound and lower bound of the program are the same.

TIME COMPLEXITY ANALYSIS

Definition 2.1

$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Definition 2.2

$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

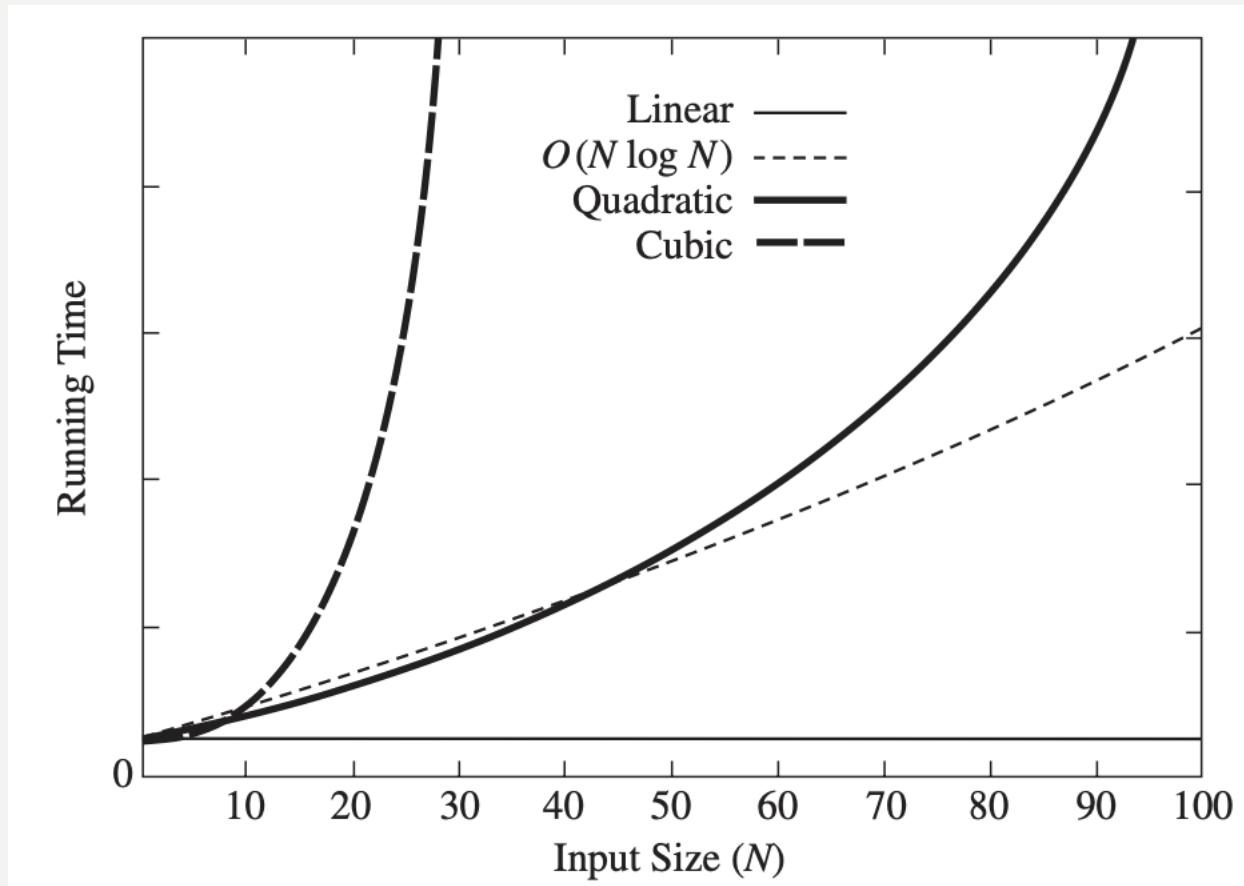
Definition 2.3

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

TIME COMPLEXITY ANALYSIS

- Four major classes of time complexity terms (from the fastest to the slowest):
 - constant
 - logarithm
 - polynomial
 - exponential
- Associated parameters (say base or exponent) does not matter between classes
 - $\log_2 n$ will be smaller than $n^{0.0001}$ as n gets larger
- We only consider the parameter when comparing terms within the same class
 - n^2 is faster than n^3

TIME COMPLEXITY ANALYSIS



TIME COMPLEXITY ANALYSIS

- The majority of algorithms we will discuss in the course will have a time complexity lower than exponential.
- In theory, a clear distinction is made between polynomial-time algorithms and exponential-time algorithms:
 - polynomial-time algorithms and faster algorithms are deemed **tractable**
 - exponential-time algorithms are deemed **intractable**
- More discussions regarding the issue in EECS 660 and EECS 764

SUMMARY

- C++ features
 - pointers and references
 - parameter passing
 - C++ objects
 - the big five: fundamental methods for data structure ADTs
 - template
 - function as parameter and function object
- Time complexity analysis
 - measures how efficient the algorithm is
 - function that converts the input size to the number of primitive operations
 - no constant coefficient, only the fastest-growing term
 - upper bound, lower bound, tight bound

EECS 560

DATA STRUCTURES

MODULE I: VECTOR AND LIST

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

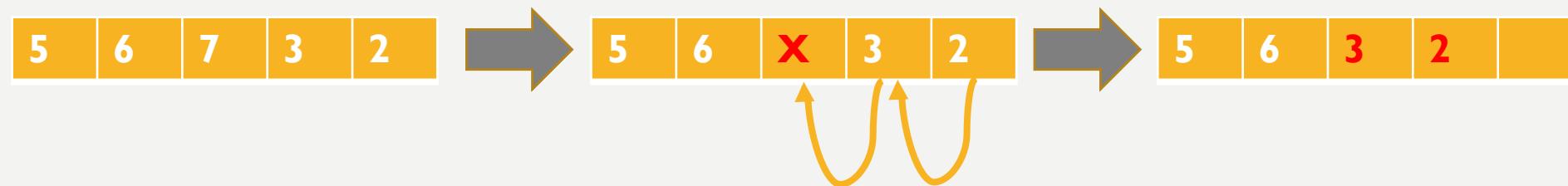
- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++, 4th edition*, by Mark Allen Weiss.

LIST

- List is an ADT that stores a list of elements
- Data
 - Array
 - Linked list
- Methods
 - Insert(x, i): inserts an element x into the i th position of the list
 - Delete(i): deletes the i th element from the list
 - Find(x, i): returns the position of the first occurrence of x in the list after i
 - At(i): returns the i th element
 - Size(): returns the number of the elements in the list
 - Reserve(n): reserves space for n elements

LIST: ARRAY

- The simplest implementation of the list ADT is to use array.
- One potential issue is that the array is fix-sized, while we may be having a large list to store that is beyond the capacity of the array.
- The second issue is that when we insert or delete an element, we may also need to move a large number of elements



LIST: ARRAY CONT.

- `Insert(x, i)`: this operation will first locate move every element after the $i-1$ th (move the n th first, followed by the $n-1$ th, ..., at last move the i th one) one position after to reserve the space for x , and insert x at the i th position. In the worst-case scenario (when $i=0$), we may need to move every element, and the time complexity is $O(n)$.
- `Delete(x,i)`: similar to `Insert()`, it also requires $O(n)$ time.
- `Find(x, i)`: we will simply scan through the array after position i until we find x . This operation costs $O(n)$ time.
- `At(i)`: because we use an array, the i th element can be immediately retrieved. This operation costs $O(1)$ time (a constant time).
- `Size()`: we can simply use a counter to store the number of elements. Updating and retrieving the counter both take $O(1)$ time.
- `Reserve(n)`: when overflow, we need to find a consecutive space of n to relocate the information. This could lead to a time complexity of $O(n)$ in the worst case scenario.
- `PrintAll()`: printing all elements of the list. Given the pre-fetch technique, this can be very efficient with array, although it still takes $O(n)$ time.

LIST: ARRAY CONT.

- Good at quickly retrieving the i th element and printing all elements
- Not so good at insertion, deletion, and information relocation

LIST: LINKED LIST

- A different data type to store a list
- Each element is implemented as a node, and the nodes are connected through pointers

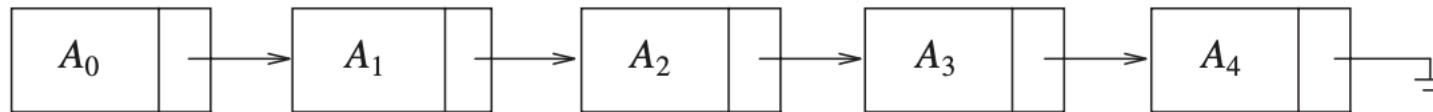


Figure 3.1 A linked list

- To facilitate bi-directional traversal, we can use double-linked list

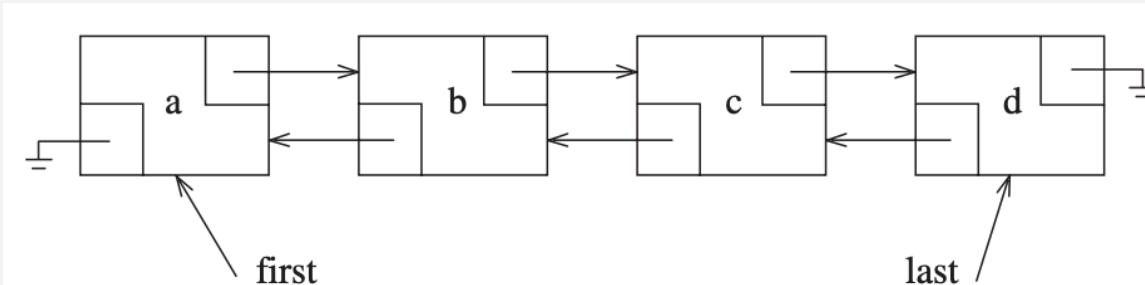


Figure 3.4 A doubly linked list

LIST: LINKED LIST CONT.

- The linked list implementation works better for insertion, deletion, and relocation.
- The main reason is that all elements no longer need to be stored in a consecutive chunk of memory space.
- However, it performs worse in retrieving the i th element and in enumerating all elements in the list.

LIST: LINKED LIST INSERTION

- Set $X \rightarrow \text{next}$ to A_2
- Set $A_1 \rightarrow \text{next}$ to X
- $O(1)$ time

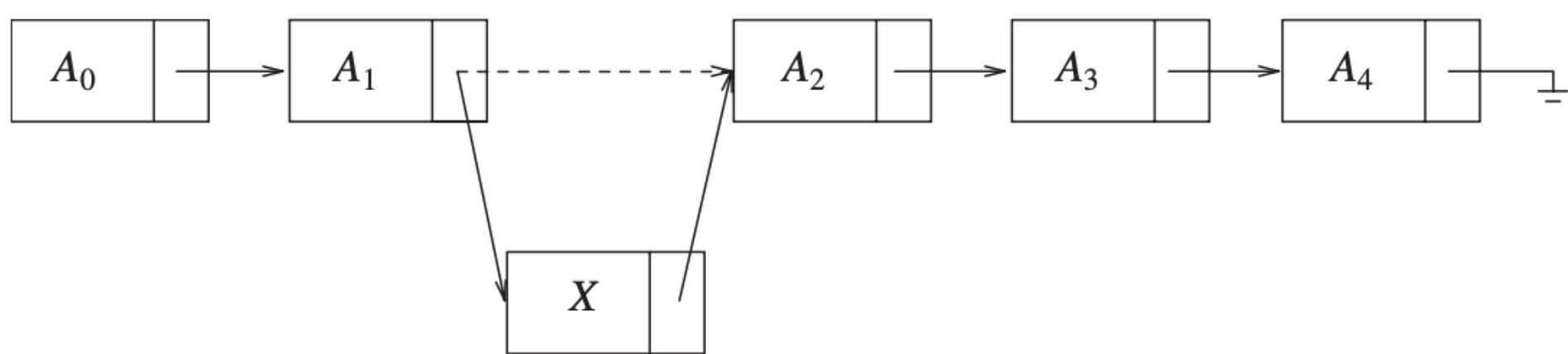


Figure 3.3 Insertion into a linked list

LIST: LINKED LIST DELETION

- Set $A_1 \rightarrow \text{next}$ to $A_2 \rightarrow \text{next}$
- Set $A_2 \rightarrow \text{next}$ to NULL
- Destruct A_2
- $O(1)$ time

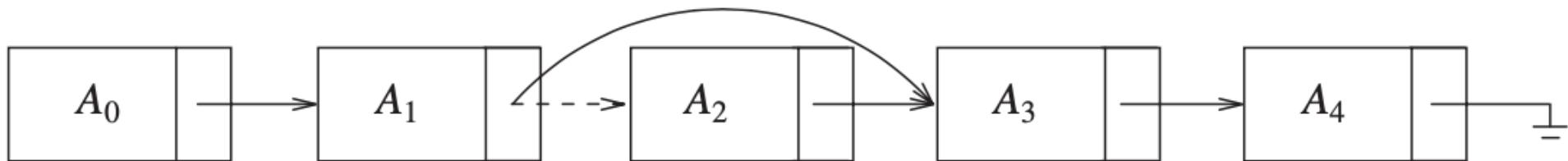


Figure 3.2 Deletion from a linked list

LIST: LINKED LIST OTHER OPERATIONS

- Two special nodes: the head (`head->prev = NULL`) and the tail (`tail->next = NULL`)
- Traversal of the entire list: start from the head and keep following the "`->next`" pointer
 - `Find()`, `At()`, `PrintAll()` will all be addressed by a traversal of the linked list, and all of them will need $O(n)$ time.
 - `Reserve()` is no longer needed for linked list, because the overflow problem does not exist.
 - `Size()` can be retrieved in $O(1)$ time if we keep a counter in the ADT (similar to the array implementation)

LIST: C++ STL VECTOR IMPLEMENTATION

- C++ STL “vector”: uses array implementation
- C++ STL “list”: uses linked list implementation

- We are going to dive into the implementational details of both ADTs

LIST: C++ STL VECTOR IMPLEMENTATION (USING ARRAY)

- These methods are to be supported by all STL containers
 - `int size() const`: returns the number of elements in the container.
 - `void clear()`: removes all elements from the container.
 - `bool empty() const`: returns true if the container contains no elements, and false otherwise.

LIST: C++ STL VECTOR IMPLEMENTATION

- These methods are supported by both the vector ADT and the list ADT
 - `void push_back(const Object & x)`: adds `x` to the end of the list.
 - `void pop_back()`: removes the object at the end of the list.
 - `const Object & back() const`: returns the object at the end of the list (a mutator that returns a reference is also provided).
 - `const Object & front() const`: returns the object at the front of the list (a mutator that returns a reference is also provided).

LIST: C++ STL VECTOR IMPLEMENTATION

- These methods are unique for the vector ADT
 - `Object & operator[] (int idx)`: returns the object at index `idx` in the `vector`, with no bounds-checking (an accessor that returns a constant reference is also provided).
 - `Object & at(int idx)`: returns the object at index `idx` in the `vector`, with bounds-checking (an accessor that returns a constant reference is also provided).
 - `int capacity() const`: returns the internal capacity of the `vector`. (See Section 3.4 for more details.)
 - `void reserve(int newCapacity)`: sets the new capacity. If a good estimate is available, it can be used to avoid expansion of the `vector`. (See Section 3.4 for more details.)

LIST: C++ STL VECTOR IMPLEMENTATION

- The data fields

```
117     private:  
118         int theSize;           // the number of elements in the list  
119         int theCapacity;        // the maximum number of elements the array can hold  
120         Object * objects;      // the array holding the elements
```

LIST: C++ STL VECTOR IMPLEMENTATION

- The constructors

```
// the explicit keyword  
// disables implicit  
// conversion and copy-  
// initialization  
// default parameter setting  
7     explicit Vector( int initSize = 0 ) : theSize{ initSize },  
8         theCapacity{ initSize + SPARE_CAPACITY }  
9     { objects = new Object[ theCapacity ]; }  
10  
11    Vector( const Vector & rhs ) : theSize{ rhs.theSize },  
12        theCapacity{ rhs.theCapacity }, objects{ nullptr }  
13    {  
14        objects = new Object[ theCapacity ];  
15        for( int k = 0; k < theSize; ++k )  
16            objects[ k ] = rhs.objects[ k ];  
17    }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- The destructor

```
26     ~Vector( )
27     { delete [ ] objects; }
28
```

LIST: C++ STL VECTOR IMPLEMENTATION

- The copy/move operation

// the double ampersand indicates a reference of a rvalue, allowing the parameter to be modified

```
19     Vector & operator= ( const Vector & rhs )
20     {
21         Vector copy = rhs;
22         std::swap( *this, copy );
23         return *this;
24     }
25
26     Vector & operator= ( Vector && rhs )
27     {
28         std::swap( theSize, rhs.theSize );
29         std::swap( theCapacity, rhs.theCapacity );
30         std::swap( objects, rhs.objects );
31         rhs.objects = nullptr;
32         rhs.theSize = 0;
33         rhs.theCapacity = 0;
34     }
35 }
```

```
36
37     Vector & operator= ( Vector && rhs )
38     {
39         std::swap( theSize, rhs.theSize );
40         std::swap( theCapacity, rhs.theCapacity );
41         std::swap( objects, rhs.objects );
42
43         return *this;
44     }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Resize and reserve

```
46     void resize( int newSize )
47     {
48         if( newSize > theCapacity )
49             reserve( newSize * 2 );
50         theSize = newSize;
51     }
```

```
53     void reserve( int newCapacity )
54     {
55         if( newCapacity < theSize )
56             return;
57
58         Object *newArray = new Object[ newCapacity ];
59         for( int k = 0; k < theSize; ++k )
60             newArray[ k ] = std::move( objects[ k ] );
61
62         theCapacity = newCapacity;
63         std::swap( objects, newArray );
64         delete [ ] newArray;
65     }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Retrieve an element

```
67     Object & operator[]( int index )
68         { return objects[ index ]; }
69     const Object & operator[]( int index ) const
70         { return objects[ index ]; }
```

- Check size and capacity

```
72     bool empty( ) const
73         { return size( ) == 0; }
74     int size( ) const
75         { return theSize; }
76     int capacity( ) const
77         { return theCapacity; }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Insert/delete/retrieve the last element

```
79     void push_back( const Object & x )
80     {
81         if( theSize == theCapacity )
82             reserve( 2 * theCapacity + 1 );
83         objects[ theSize++ ] = x;
84     }
85
86     void push_back( Object && x )
87     {
88         if( theSize == theCapacity )
89             reserve( 2 * theCapacity + 1 );
90         objects[ theSize++ ] = std::move( x );
91     }
```

```
93     void pop_back( )
94     {
95         --theSize;
96     }
97
98     const Object & back( ) const
99     {
100         return objects[ theSize - 1 ];
101     }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Return iterators

```
106     iterator begin( )
107     { return &objects[ 0 ]; }
108     const_iterator begin( ) const
109     { return &objects[ 0 ]; }

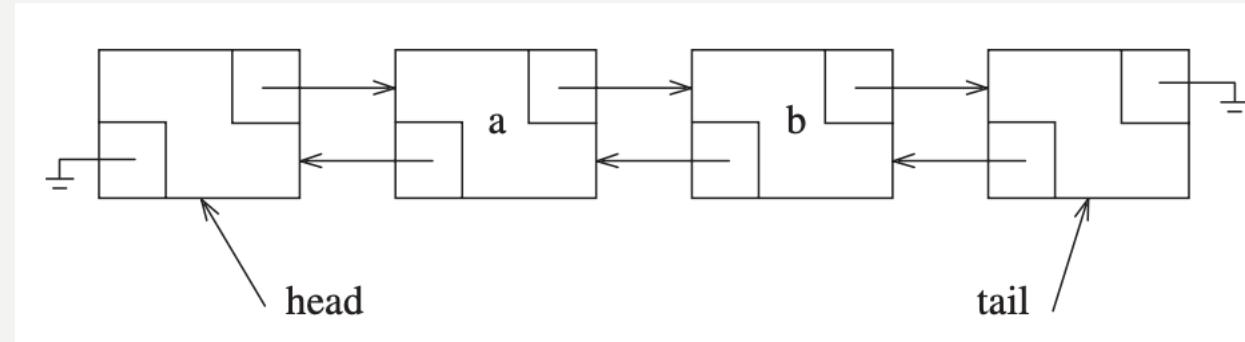
110    iterator end( )
111    { return &objects[ size( ) ]; }
112    const_iterator end( ) const
113    { return &objects[ size( ) ]; }
```

LIST: C++ STL LIST IMPLEMENTATION

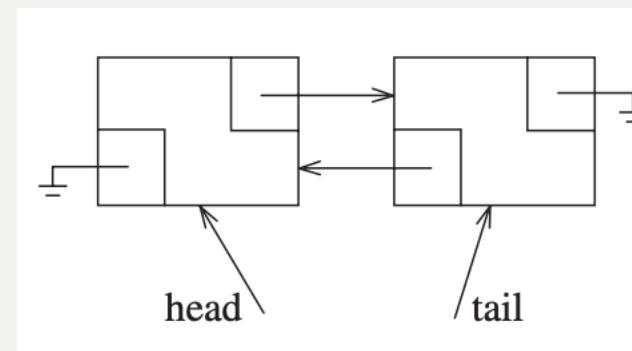
- These methods are uniquely available in list
 - `void push_front(const Object & x)`: adds x to the front of the list.
 - `void pop_front()`: removes the object at the front of the list.

LIST: C++ STL LIST IMPLEMENTATION

- The head and the tail



- An empty list (which also contains the head and tail)



LIST: C++ STL LIST IMPLEMENTATION

- The node definition

```
1  struct Node
2  {
3      Object data;
4      Node *prev;    // pointer to the previous node
5      Node *next;    // pointer to the next node
6
7      Node( const Object & d = Object{ }, Node * p = nullptr,
8             Node * n = nullptr )
9          : data{ d }, prev{ p }, next{ n } { }
10
11     Node( Object && d, Node * p = nullptr, Node * n = nullptr )
12         : data{ std::move( d ) }, prev{ p }, next{ n } { }
13 }
```

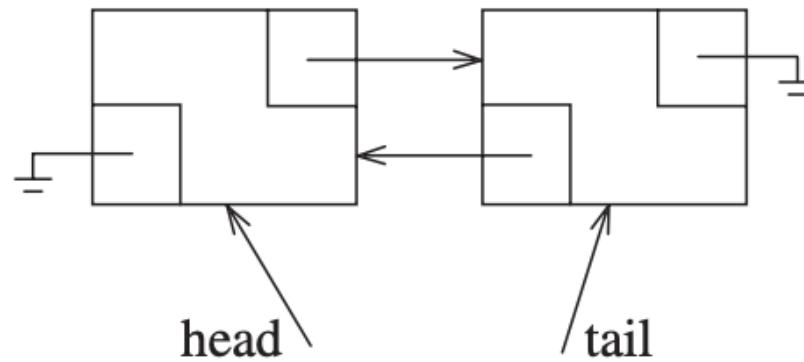
LIST: C++ STL LIST IMPLEMENTATION

- list definition and initialization

```
79     private:  
80         int    theSize;  
81         Node *head;  
82         Node *tail;
```

// theSize: counter for the
number of elements in the list

```
43     void init( )  
44     {  
45         theSize = 0;  
46         head = new Node;  
47         tail = new Node;  
48         head->next = tail;  
49         tail->prev = head;  
50     }
```



LIST: C++ STL LIST IMPLEMENTATION

- constructors and destructor

```
1      List( )
2          { init( ); }
3
4      ~List( )
5      {
6          clear( );
7          delete head;
8          delete tail;
9      }
10
11     List( const List & rhs )
12     {
13         init( );
14         for( auto & x : rhs )
15             push_back( x );
16     }
```

LIST: C++ STL LIST IMPLEMENTATION

- Copy/assignment

```
18     List & operator= ( const List & rhs )
19     {
20         List copy = rhs;
21         std::swap( *this, copy );
22         return *this;
23     }
24
25
26     List( List && rhs )
27         : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
28     {
29         rhs.theSize = 0;
30         rhs.head = nullptr;
31         rhs.tail = nullptr;
32     }
```

LIST: C++ STL LIST IMPLEMENTATION

- `const_iterator` internal definition

```
28     protected:  
29         Node *current;  
30  
31         Object & retrieve( ) const  
32             { return current->data; }  
33  
34         const_iterator( Node *p ) : current{ p }  
35             { }  
36  
37         friend class List<Object>; // “friend class” allows to access  
// the private and protected  
// members of List<Object>
```

LIST: C++ STL LIST IMPLEMENTATION

- `const_iterator` continued

```
4     const_iterator( ) : current{ nullptr }
5     {
6
7         const Object & operator* ( ) const
8             { return retrieve( ); }
9
10        const_iterator & operator++ ( )
11        {
12            current = current->next;
13            return *this;
14        }
15
16        const_iterator operator++ ( int )
17        {
18            const_iterator old = *this;
19            +( *this );
20            return old;
21        }
```

```
23        bool operator== ( const const_iterator & rhs ) const
24            { return current == rhs.current; }
25        bool operator!= ( const const_iterator & rhs ) const
26            { return !( *this == rhs ); }
```

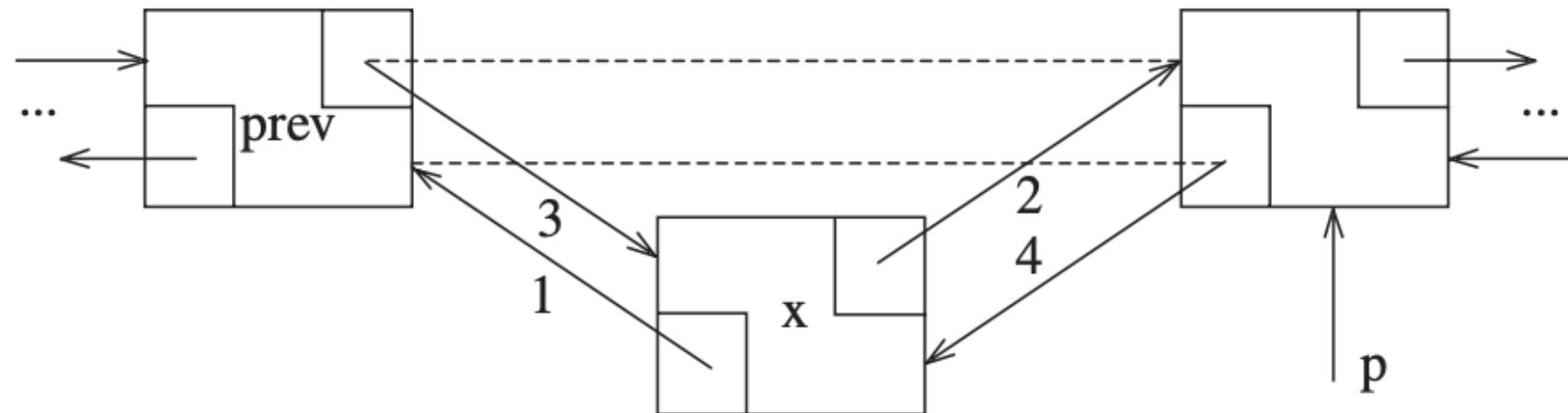
LIST: C++ STL LIST IMPLEMENTATION

- iterator

```
42         iterator( )
43             {
44
45             Object & operator* ( )
46                 { return const_iterator::retrieve( ); }
47             const Object & operator* ( ) const
48                 { return const_iterator::operator*( ); }
49
50             iterator & operator++ ( )
51             {
52                 this->current = this->current->next;
53                 return *this;
54             }
55
56             iterator operator++ ( int )
57             {
58                 iterator old = *this;
59                 +( *this );
60                 return old;
61             }
62
63     protected:
64         iterator( Node *p ) : const_iterator{ p }
65             {
66
67     friend class List<Object>;
```

LIST: C++ STL LIST IMPLEMENTATION

- insertion



```
Node *newNode = new Node{ x, p->prev, p }; // Steps 1 and 2  
p->prev = p->prev->next = newNode; // Steps 3 and 4
```

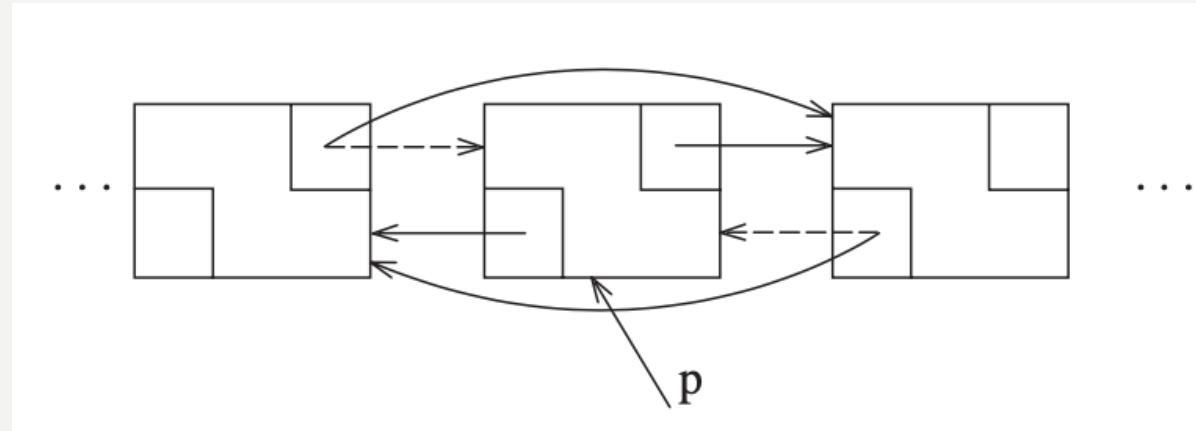
LIST: C++ STL LIST IMPLEMENTATION

- insertion continued

```
1      // Insert x before itr.  
2      iterator insert( iterator itr, const Object & x )  
3      {  
4          Node *p = itr.current;  
5          theSize++;  
6          return { p->prev = p->prev->next = new Node{ x, p->prev, p } };  
7      }  
8  
9      // Insert x before itr.  
10     iterator insert( iterator itr, Object && x )  
11     {  
12         Node *p = itr.current;  
13         theSize++;  
14         return { p->prev = p->prev->next  
15                         = new Node{ std::move( x ), p->prev, p } };  
16     }
```

LIST: C++ STL LIST IMPLEMENTATION

- deletion



```
p->prev->next = p->next;  
p->next->prev = p->prev;  
delete p;
```

LIST: C++ STL LIST IMPLEMENTATION

- deletion continued

```
1      // Erase item at itr.
2      iterator erase( iterator itr )
3      {
4          Node *p = itr.current;
5          iterator retVal{ p->next };
6          p->prev->next = p->next;
7          p->next->prev = p->prev;
8          delete p;
9          theSize--;
10
11         return retVal;
12     }
13
14     iterator erase( iterator from, iterator to )
15     {
16         for( iterator itr = from; itr != to; )
17             itr = erase( itr );
18
19         return to;
20     }
```

SUMMARY

- the list ADT can be used to represent a list of ordered objects
- list can be implemented using array (STL vector) or linked list (STL list)
- Lab 1: implementation of vector
- Lab 2: implementation of linked list

EECS 560

DATA STRUCTURES

MODULE II: QUEUE AND STACK

DISCLAIMER

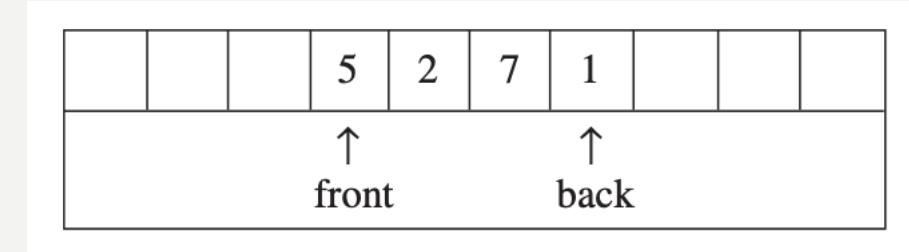
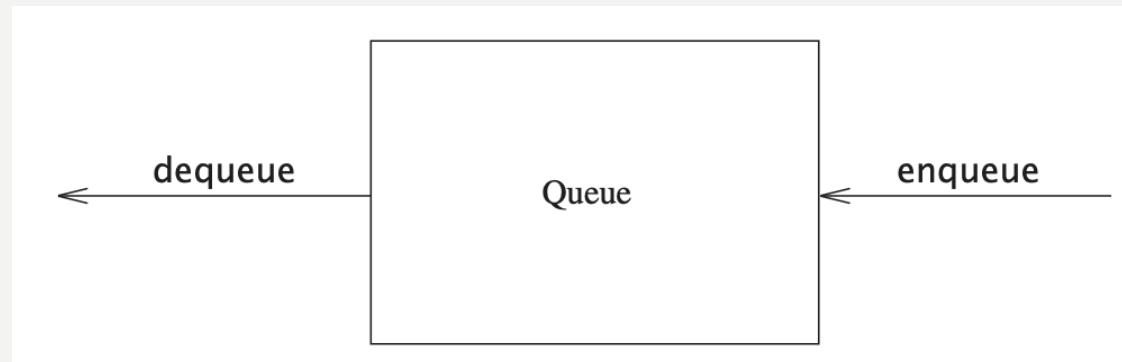
- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++, 4th edition*, by Mark Allen Weiss.

QUEUE

- Queue is an extension of the list ADT that has the “first in first out” behavior (while the other operations remain the same as previously discussed)
- Two essential operations of queue are:
 - enqueue (or push): insert an element into the list at the last position (also called “back”)
 - dequeue: remove the first element (also called “front”) of the list (its content is also returned)



QUEUE APPLICATIONS

- It can serve as a “buffer” that resembles a real queue
 - Imagine you are waiting in a line for checkout
 - the line can be modeled as a queue
 - the first arriving customer goes to checkout first, representing the “first in first out” model
- More applications in graph traversal (discussed later in the graph ADT module)

QUEUE IMPLEMENTATION (LINKED LIST)

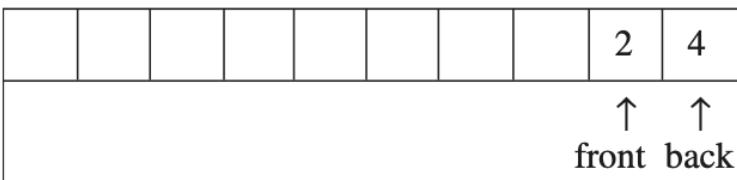
- The implementation is straightforward (and we will do that as a lab)
 - enqueue(x): this can be implemented as insert(x, theSize), i.e., inserting x at the last position
 - dequeue(): this can be implemented as delete(1), i.e., deleting the element before the second element
 - both operations clearly take $O(1)$ time

QUEUE IMPLEMENTATION (ARRAY)

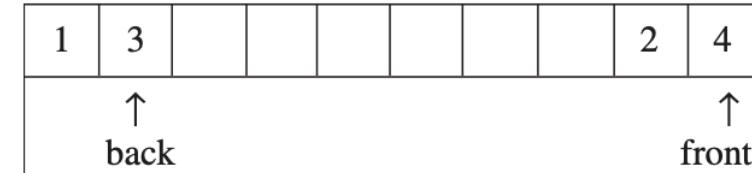
- The implementation will work on a fix-sized array
- We will use two pointers (front and back) to mark the boundaries of the list
 - enqueue: move forward (towards the right) the back pointer
 - dequeue: move forward (towards the right) the front pointer
 - note that the two pointers do not necessarily point to the starting and ending positions of the array

QUEUE IMPLEMENTATION (ARRAY)

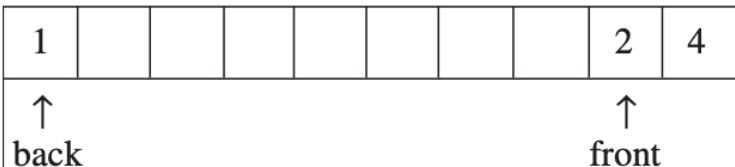
Initial state



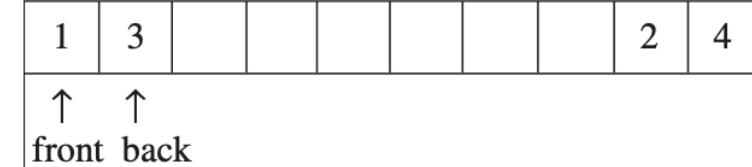
After dequeue, which returns 2



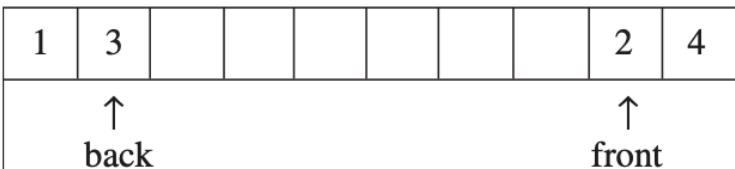
After enqueue(1)



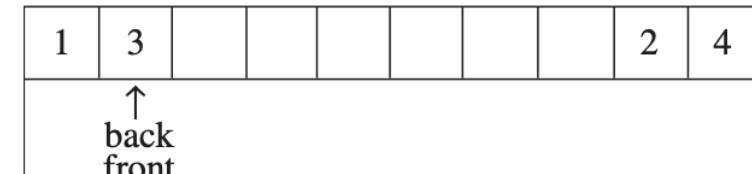
After dequeue, which returns 4



After enqueue(3)

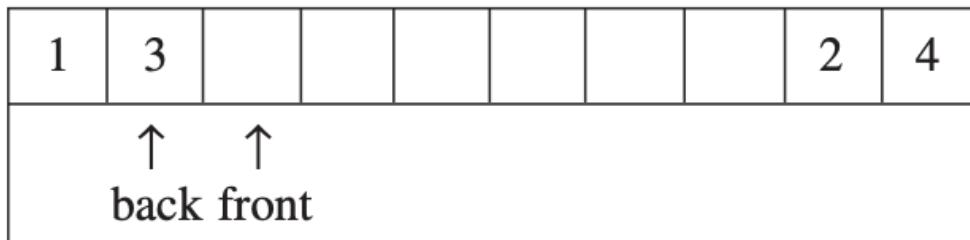


After dequeue, which returns 1



QUEUE IMPLEMENTATION (ARRAY)

After dequeue, which returns 3
and makes the queue empty



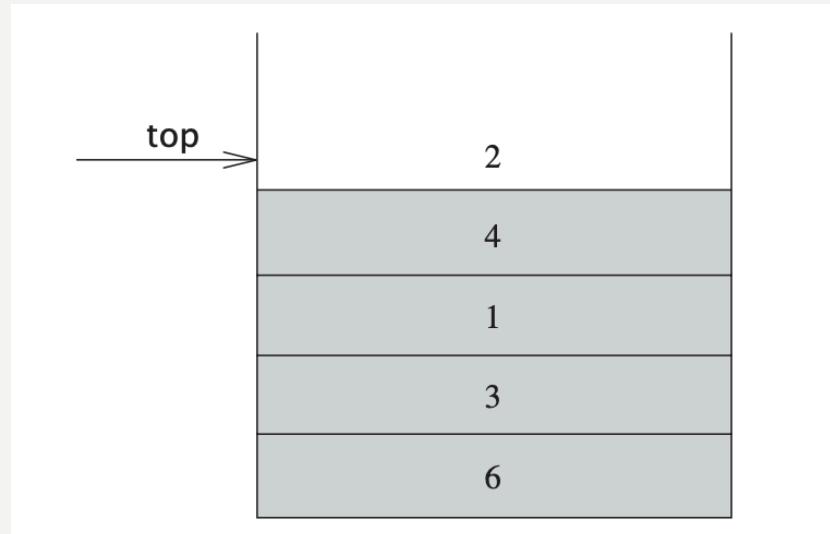
- Note: this also indicates the initial state when the list is empty. And we can represent an empty list by setting the front pointer one position after the back pointer (which position the back pointer is pointing at does not matter).

QUEUE IMPLEMENTATION (ARRAY)

- When resizing the array, we need to copy all elements in the current array (between the front and the back pointers, inclusively) to the new array.
- Other STL supported queue operations:
 - `front()`: returns the value pointed by the front pointer (without modifying it)
 - `back()`: returns the value pointed by the back pointer (without modifying it)

STACK

- Stack is also an extension of the list ADT.
- Unlike the queue's "first in first out" behavior, the stack supports the "first in last out"(or equivalently "last in first out") operations
 - push: inserting an element to the last position of the list
 - pop: deleting the last element in the list (and also return its value)



STACK APPLICATIONS

- Arithmetic computation
 - helps the computer parse arithmetic expressions to determine the correct execution order of different operators
- More applications in graph traversal (discussed later in the graph ADT module)

STACK APPLICATIONS

- Consider the following example. If we simply compute the operations from left to right, we are likely getting a wrong answer of 19.37

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

5.29
11.28
18.27
19.37

- But we know the correct execution order should be $(4.99 * 1.06) + 5.99 + (6.99 * 1.06) = 18.69$

STACK APPLICATION

- The computer uses stack to determine the correct computation order, according to the priorities of different operators in two steps
 - first, convert the infix expression (what we regularly write) into the postfix expression
 - second, compute the value using the postfix expression
- For simplicity, we assume only three operations
 - (): parenthesis, which has the highest priority
 - *: multiplication, which has a lower priority
 - +: addition, which has the lowest priority
 - the minus operation can be viewed as adding the corresponding negative operand, while the division operation can be viewed as multiplying the corresponding reciprocal operand

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

- Consider the following infix expression

$$a + b * c + (d * e + f) * g$$

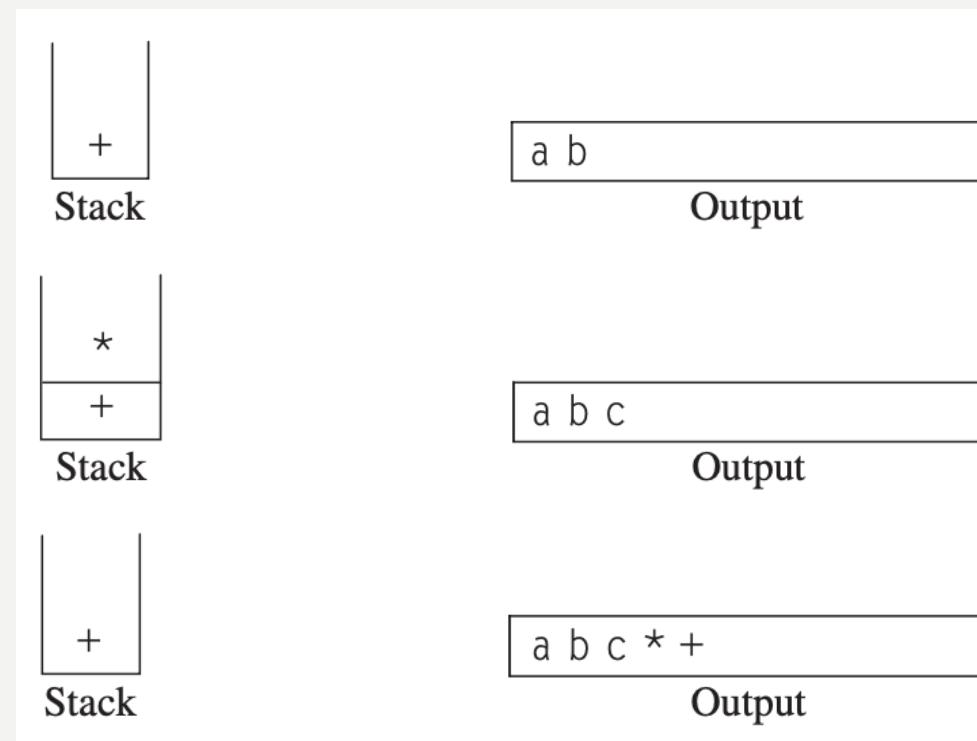
- We will convert it to the postfix form by scanning the expression from left to right, and apply the following rules:
 - if we see an operand, we output it
 - if we see an operator, we pop all operators in the stack that have the same or higher priorities than it and write them to the output. After that we push the current operator into the stack
 - exception: if the operator is a right parenthesis, we pop every operator until we see a left parenthesis (which will also be popped)
 - when all symbols are read, pop all symbols in the stack out and write to the output

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

- Why would the algorithm work?
 - we can interpret the postfix expression as an ordered execution of a set of operations. That is, the earlier we see an operator in the postfix expression, the earlier we execute the operation.
 - so, when we pop the operators, we ensure that the operators that are being popped have the same or higher priority than then current operator. It indicates that we allow operators that with higher priorities to execute earlier.
 - we will go through a concrete example

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

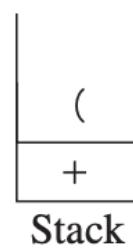
a + b * c + (d * e + f) * g



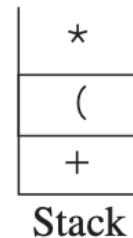
// when we see the third operator “+”, we pop the second operator “*” and the first operator “+” out because they have the same or higher priority than “+”

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

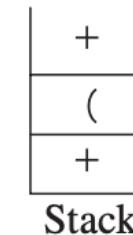
a + b * c + (d * e + f) * g



a b c * + d
Output



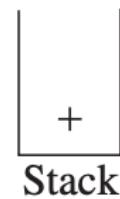
a b c * + d e
Output



a b c * + d e * f
Output

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

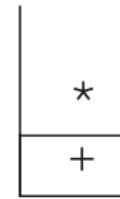
a + b * c + (d * e + f) * g



Stack

a b c * + d e * f +

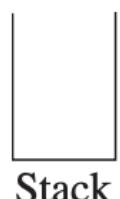
Output



Stack

a b c * + d e * f + g

Output



Stack

a b c * + d e * f + g * +

Output

STACK APPLICATION (POSTFIX COMPUTATION)

- Now we have converted the infix expression to postfix expression
 - infix: $a+b*c+(d*e+f)*g$
 - postfix: $abc^*+de^*f+g^*$
- The next step is to compute the actual value using the postfix expression using the following steps (also involve stack)
 - we scan the postfix expression from left to right
 - if we see an operand, we push it to the stack
 - if we see an operator, we pop two operands from the stack and perform the corresponding operation, and push the resulted number back to the stack

STACK APPLICATION (POSTFIX COMPUTATION)

- consider the postfix expression we just got: $abc^*+de^*f+g^*$

		c			e		f		g			
b	b	b	(b^*c)		d	d	(d^*e)	(d^*e)	$(d^*e)+f$	$(d^*e)+f$	$((d^*e)+f)^*g$	
a	a	a	a	$a+(b^*c)$								

And finally we get: $(a+(b^*c))+((d^*e)+f)^*g$

STACK APPLICATION (POSTFIX COMPUTATION)

- compare the results from parsing postfix expression and the original infix expression
 - postfix parsing: $(a+(b*c))+((d*e)+f)*g$
 - infix: $a+b*c+(d*e+f)*g$
- We can easily confirm that the computation is correct

SUMMARY

- Queue and stack are extensions of the original list data structure
 - queue: first in first out
 - stack: last in first out
- Many important applications, and are considered as two fundamental data structures

EECS 560

DATA STRUCTURES

MODULE III: HASH TABLE

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++*, 4th edition, by Mark Allen Weiss.

HASH TABLE

- Hash table is an ADT that supports fast (ideally $O(1)$ time) search of an item and retrieve its related information
 - For example, we have registered all your favorite video games
 - We will be using a data structure to address the following need: given the name of the student, retrieve her/his favorite video game

Name	Game
Alex	Warcraft
Mary	Minecraft
Lisa	Candy Crush Saga
Joy	Angry Bird
Tom	Call of Duty

HASH TABLE

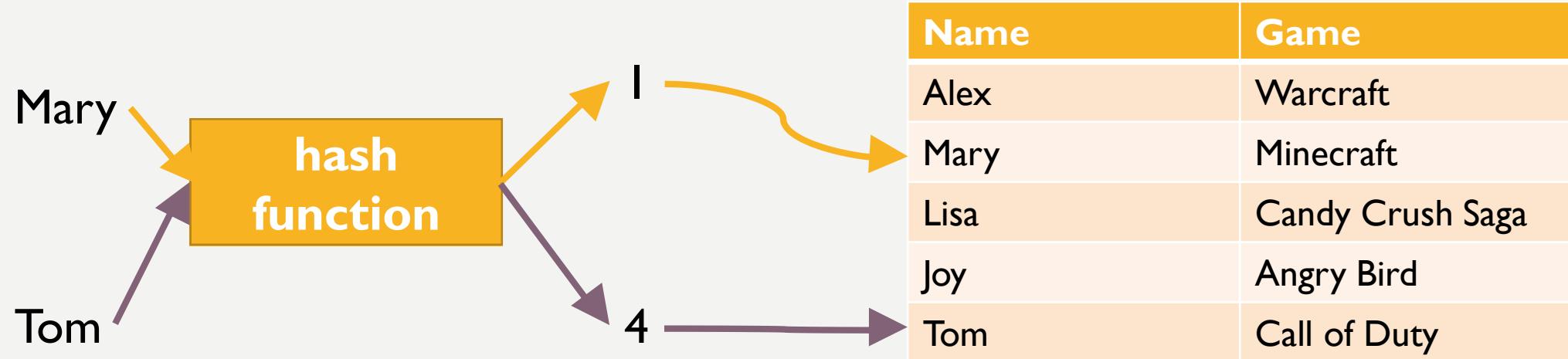
- The simplest solution is to go through the array, and find the name of the student to retrieve the name of the game
- However, in the worst case it will take $O(n)$ time, if we have n students
- Or, if we have spent extra time to sort the records alphabetically, we can use binary search to look for the name, and it will take $O(\log(n))$ time. It is faster but still not constant.

HASH TABLE

- In the context of hash table
 - The item we use to perform the search is called “key” (i.e., the name of the student). We assume all keys are unique (e.g., no two students have identical names).
 - The information we are searching for is called “value” (i.e., the name of the video game). Duplicated values are allowed (e.g., two students may love the same video game, that’s OK).
 - Each hash table record is in fact a key-value pair.
 - The records are stored in a fix-sized array. That is, the number of key-value pairs we can store in a given hash table is fixed (unless we perform some resizing operations).

HASH TABLE

- Instead of directly searching for the key, the hash table ADT introduces a function, called hash function, to map each key to the position of its corresponding record in the hash table.
- Computing the hash function should only take $O(1)$ time, given a fix-sized key size. In this case the entire retrieval process will take $O(1)$ time.



HASH TABLE

- There are three fundamental challenges in implementing a hash table:
 - **The choice of hash function:** Ideally, we should choose hash functions that can distribute the records throughout the array as evenly as possible. If two records are assigned to the same position of the array (called hash collision, or simply collision), extra efforts are needed to retrieve the correct information (discussed later).
 - **The handling of collision:** Practically, there is no perfect hash function that can completely avoid collision. We expect ways to handle collision with high efficiency and accuracy.
 - **The handling of array overflow:** The hash table is implemented using a fix-sized array. When more records are added and causes the overflow, we will need to resize and rehash all elements.

HASH TABLE: HASH FUNCTION

- The main objectives of designing a hash function:
 - evenly distribute the records
 - easy to compute (associate with a smaller constant even in $O(1)$ time)
- Practically, the hash function expects the key as a numerical value
 - if it is not, we can convert it into a numerical value (e.g., using ASCII to convert string, or using binary to represent other more complicated objects)

HASH TABLE: HASH FUNCTION

- Simple hash function: taking the modular of the array size

```
1 int hash( const string & key, int tableSize )
2 {
3     int hashVal = 0;
4
5     for( char ch : key )
6         hashVal += ch;
7
8     return hashVal % tableSize;
9 }
```

- Caveats:
 - cannot fully distribute all records if the number of possible records is smaller than the table size, yet will cause collision if we use a small table size
 - try setting the table size to a prime number to avoid unexpected collision

HASH TABLE: HASH FUNCTION

- Solution: extending the range we can map the key

```
1  /**
2   * A hash routine for string objects.
3   */
4  unsigned int hash( const string & key, int tableSize )
5  {
6      unsigned int hashVal = 0;
7
8      for( char ch : key )
9          hashVal = 37 * hashVal + ch;
10
11     return hashVal % tableSize;
12 }
```

HASH TABLE: HASH FUNCTION

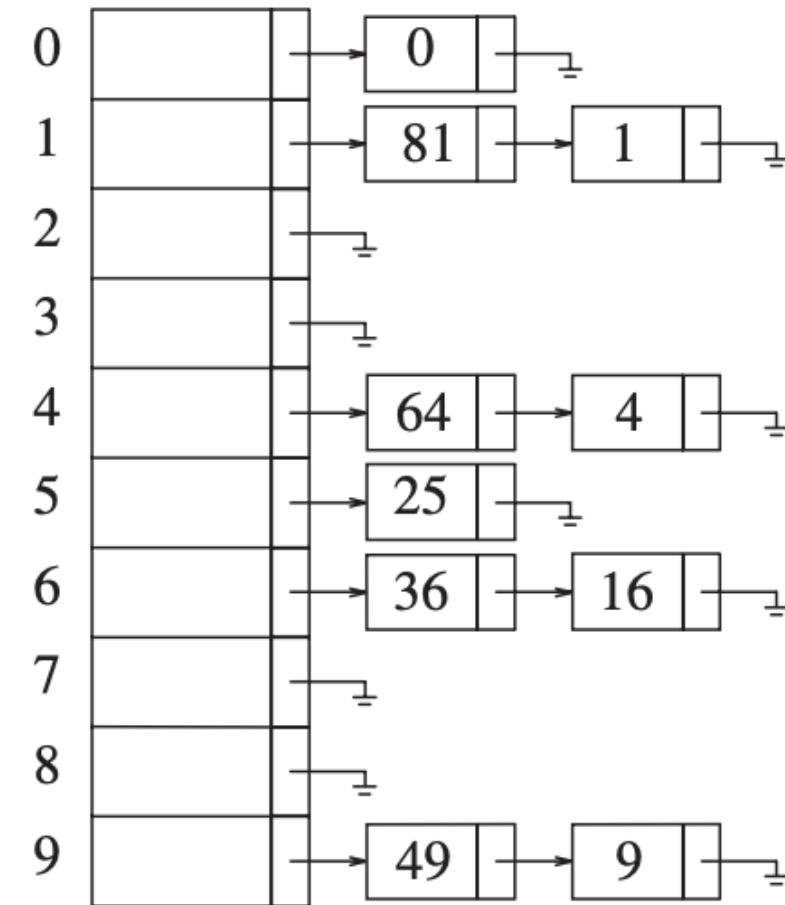
- Designing a good hash function is an art.
- A good hash function considers the features of the keys (such as the distribution of the values to avoid collision).

HASH TABLE: HANDLING COLLISION

- Collision happens when we have two different records that are hashed into the same position of the array
- The solution is intuitive:
 - either we append the record into the current position (separate chaining)
 - or we hash the record to another position that is empty (hashing without chaining)
- Either choice would increase the worst-case time complexity for inserting a record to and retrieving a record from the hash table.

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- Each position of the array is extended from a simple data holder to the head of a linked list
- When collision happens, we append the new record at the end of the linked list
- When we retrieve a record, we go through each element of the linked list ($O(n)$ time worst-case scenario)



HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- insertion implementation

```
bool insert( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ]; // computes the position using the hash function “myhash”
    if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
        return false; // error checking
    whichList.push_back( x ); // insert the element to the end of the linked list

    // Rehash; see Section 5.5
    if( ++currentSize > theLists.size( ) )
        rehash( );

    return true;
}
```

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- remove operation (we can implement the look-up operation in a similar way)

```
bool remove( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ]; // computes the position using the hash function “myhash”
    auto itr = find( begin( whichList ), end( whichList ), x );
                                                // goes through the linked list to search for the key
    if( itr == end( whichList ) )
        return false;

    whichList.erase( itr );
    --currentSize;
    return true;
}
```

// when implementing the look-up operation, we
can simply return the value

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- Disadvantages:
 - it needs an additional implementation of the linked list ADT
 - the linked list implementation requires more space than the array implementation
 - for each data, linked list requires two pointers to link the former and later data blocks
- Can we avoid the linked list?
 - we can use an alternative hash function to find another empty slot when collision happens

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- The alternative hash function is based on the primary hash function, but added with another function that depends on the number of trials.
- The quadratic probing hash function
 - $h_i(x) = (\text{hash}(x) + f(i)) \text{ (mod TableSize)}$
 - $f(i) = i^2$

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- imagine we are inserting 89, 18, 49, 58, 69 into an empty hash table

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2						
3					58	58
4						69
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

A diagram illustrating the insertion process using quadratic probing. An orange arrow labeled '+4' points from index 4 to index 8, indicating the probe sequence for the value 58. Another orange arrow labeled '+1' points from index 8 to index 9, indicating the probe sequence for the value 69.

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- Do we always find an empty slot?
- When we have a large hash table, and its size is prime, then we can expect such a nice property.
- **Theorem:** If quadratic probing is used, and the table size is prime, then the new element can always be inserted if the table is at least half empty.

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- **Proof** (using prove by contradiction):
 - Assume that the table size is a prime, and it is larger than 3
 - Consider two different hash trails i and j , where i and j are integers. We have $i \neq j$, and $0 \leq i, j \leq \text{floor}(\text{TableSize}/2)$ (this is because $i^2 > \text{TableSize}$ and $i > \text{TableSize} \geq 3$)
 - Now, if the i th and the j th probing are pointing to the same position (which means we are not able to find an empty slot), we have:
 - $(\text{hash}(x) + i^2)(\text{mod TableSize}) = (\text{hash}(x) + j^2)(\text{mod TableSize})$
 - $i^2(\text{mod TableSize}) = j^2(\text{mod TableSize})$
 - $(i^2 - j^2)(\text{mod TableSize}) = 0$
 - $(i + j)(i - j)(\text{mod TableSize}) = 0$

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- Proof cont.
 - First, $(i + j)(\text{mod } \text{TableSize})$ cannot be 0 because $i, j \leq \text{floor}(\text{TableSize}/2)$
 - Second, $(i - j)(\text{mod } \text{TableSize})$ cannot be 0 because $i, j \leq \text{floor}(\text{TableSize}/2)$ and $i \neq j$
 - Therefore, it is impossible for two different hash probes to point to the same location, if all premises are met

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: the findPos function for finding the next available position

```
int findPos( const HashedObj & x ) const
{
    int offset = 1;
    int currentPos = myhash( x );

    while( array[ currentPos ].info != EMPTY &&
           array[ currentPos ].element != x )
    {
        currentPos += offset; // Compute ith probe
        offset += 2;          // increasing the jump size for each probing failure
        if( currentPos >= array.size( ) )      // computing the modular
            currentPos -= array.size( );
    }

    return currentPos;
}
```

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: insertion

```
bool insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x );    // calling the findPos function to locate an empty slot
    if( isActive( currentPos ) )
        return false;

    array[ currentPos ].element = x;
    array[ currentPos ].info = ACTIVE;

    // Rehash; see Section 5.5
    if( ++currentSize > array.size( ) / 2 )
        rehash( );

    return true;
}
```

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: removal (similarly, look-up)

```
bool remove( const HashedObj & x )
{
    int currentPos = findPos( x );      // calling the findPos function to locate an empty slot
    if( !isActive( currentPos ) )
        return false;

    array[ currentPos ].info = DELETED;
    return true;
}
```

HASH TABLE: HANDLING COLLISION

- Other methods share a similar idea, but use different $f(i)$ functions
 - quadratic probing uses $f(i) = i^2$
 - linear probing uses $f(i) = i$
 - double hashing uses $f(i) = i * \text{hash}_2(x)$, where $\text{hash}_2(x)$ is a different hash function than the primary hash function

HASH TABLE: HANDLING OVERFLOW

- The final issue remains unsolved is how do we handle hash table overflow
 - we know that the hash table has a fixed size, while the number of records we insert is unknown and can be much larger than the pre-set table size
 - even we use separate hashing, not increasing the array size can lead to very long chains, resulting in slower insertion and look-ups
 - furthermore, for quadratic probing, we need to ensure that the table is at least half empty to make sure that we can always find an empty slot
- We should not allocate a huge space for the hash table, because it may use unnecessarily amount of memory
- Solution: we dynamically change the table size, according to the number of records in it

HASH TABLE: HANDLING OVERFLOW

- We can adopt the following strategy:
 - denote the ratio between the number of records in the hash table and the size of the hash table as the load factor
 - when we insert, if the load factor is $> 1/2$, we double the size (to a near prime number) of the hash table (called table doubling)
 - when we delete, if the load factor is $< 1/8$, we halve the size (to a near prime number) of the hash table (called table halving)

HASH TABLE: HANDLING OVERFLOW

- After a resize operation, TableSize has been changed.
- Therefore, it is necessary to revise the hash function (especially if we are taking modular of TableSize).
- This process is called rehashing.

HASH TABLE: HANDLING OVERFLOW (REHASHING)

All records need to be rehashed, which would take $O(n)$ time where n is the number of existing records

0	6
1	15
2	23
3	24
4	
5	
6	13

$\text{mod}(7)$



0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

$\text{mod}(17)$

HASH TABLE: HANDLING OVERFLOW (REHASHING)

- Fortunately, not every insertion or deletion will trigger a table resizing operation
 - Note that after a table doubling, the new load factor becomes $1/4$. It means we can delete additional $\text{TableSize}/8$ records or insert additional $\text{TableSize}/4$ records without triggering a resize operation
 - Similarly, after a table halving, the new load factor becomes $1/4$. It means we can delete additional $\text{TableSize}/8$ records or insert additional $\text{TableSize}/4$ records without triggering a resize operation
- On average, we still have $O(1)$ insertion and deletion time complexity with the implementation of the table doubling and table halving mechanism
 - proof will be discussed in EECS 660

THE C++ STL FOR HASH TABLE

- std::unordered_map and std::unordered_set
 - std::unordered_map stores both of the key and value of the records, while std::unordered_set only stores the key
 - e.g., std::unordered_map can answer questions like “What is the favorite video game of Mary?”, while std::unordered_set can only answer the question of “Is Mary one of the students in our class?”
- std::map and std::set
 - they are similar to the unordered version, but the keys in them are ordered
 - faster when you perform in-order traversal, but slower for each insertion/deletion/look-up
 - not implemented using hash table, but with tree ADTs (will be discussed more later)

SUMMARY

- Hash table allows you to quickly ($O(1)$ time) look-up entries using the keys
- Challenges in designing hash table
 - good hash function that evenly distribute the records
 - handling collisions
 - separate hashing using linked list
 - quadratic probing without linked list
 - handling overflow
 - table doubling and table halving
- C++ STL: `std::unordered_map` and `std::unordered_set`

EECS 560

DATA STRUCTURES

MODULE IV: TREE

DISCLAIMER

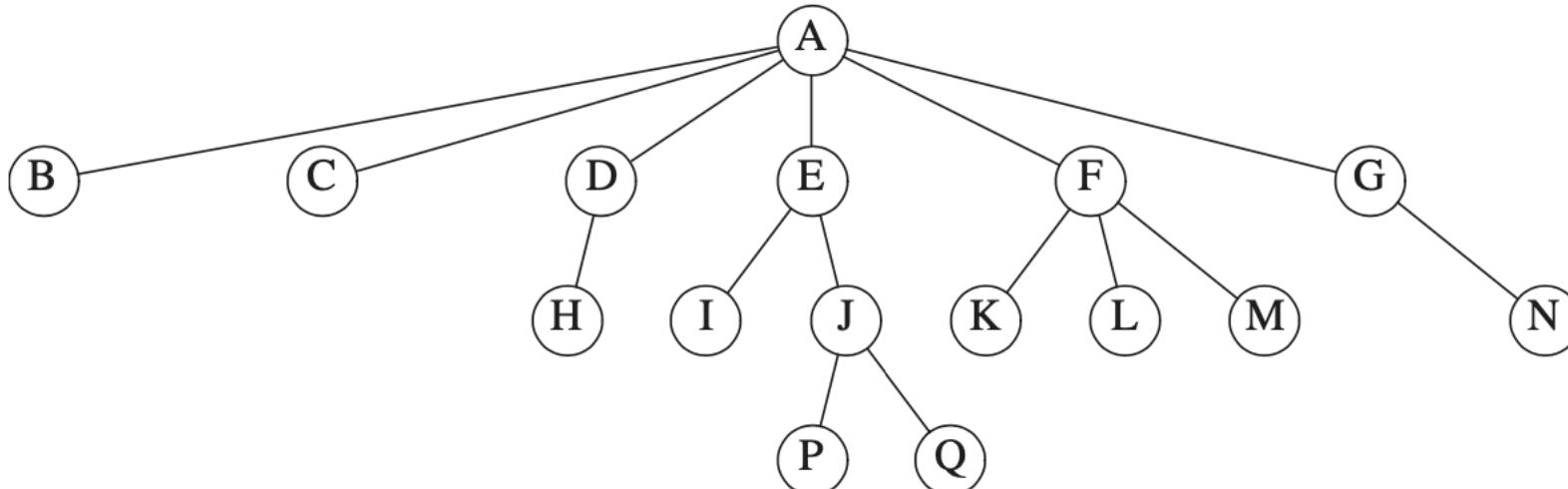
- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++, 4th edition*, by Mark Allen Weiss.

TREE

- We can think of the tree ADT as an extended linked list:
 - in the linked list, each element can link to at most one previous element and at most one next element
 - in the tree, each element can linked to at most one previous element **but multiple next elements.**



TERMINOLOGY

- In the context of tree ADT, we define the following terminology
 - each element (which stores the actual data) in the linked-list counterpart is formally defined as a tree node, or simply a **node**
 - if a node is not linked to any previous node, the node is called a **root**; if a node is not linked to any next node, the node is called a **leaf**; if a node is linked to both previous and next nodes, it is called an **internal node**
 - the between-node pointers as in the linked-list counterpart is defined as an **edge**
 - a set of ordered and continuous (the end node of the former edge is the starting node of the later edge) edges are called a **path**
 - given a node, the length (number of involved edges) of the path that goes from the root to itself is called its **depth**; the length of the longest path that goes from it to a leaf is called its **height**

TERMINOLOGY

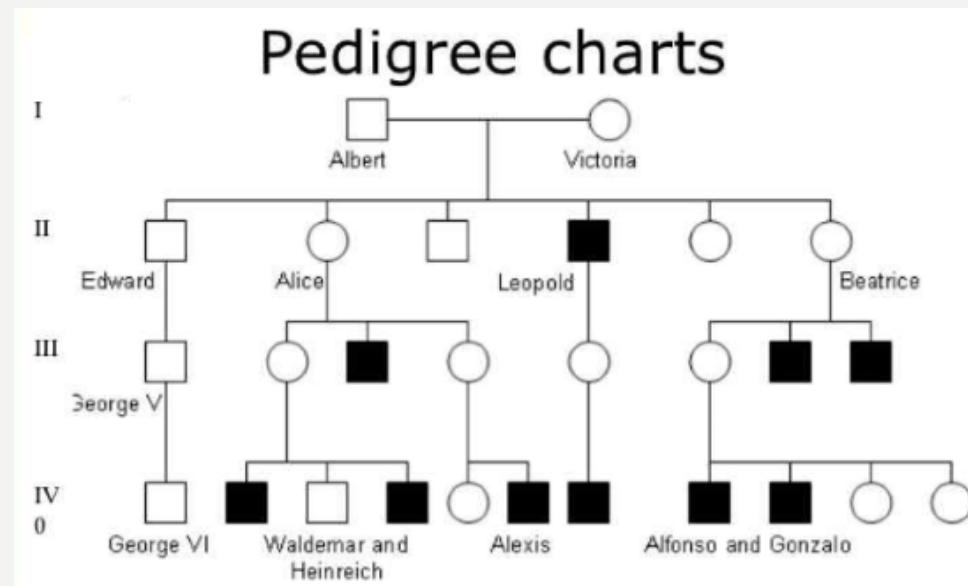
- Definition continued:
 - given a node, its immediate previous node is called its **parent**,
 - given a node, its immediate next nodes are called its **children** (or **child** if we are referring to a singular next node)
 - nodes that have the same parent are called **siblings**
 - the grandparent or grand-grandparent or grand-grand-grandparent... of a node is called its **ancestor** (alternatively, there exists a path that goes from the ancestor to the current node)
 - the grandchild or grand-grandchild or grand-grand-grandchild... of a node is called its **descendant** (alternatively, there exists a path that goes from the current node to the descendant)

DEFINITION

- A recursive definition:
 - A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished root, and zero or more nonempty (sub)trees, each of whose roots are connected by a directed edge from the root.

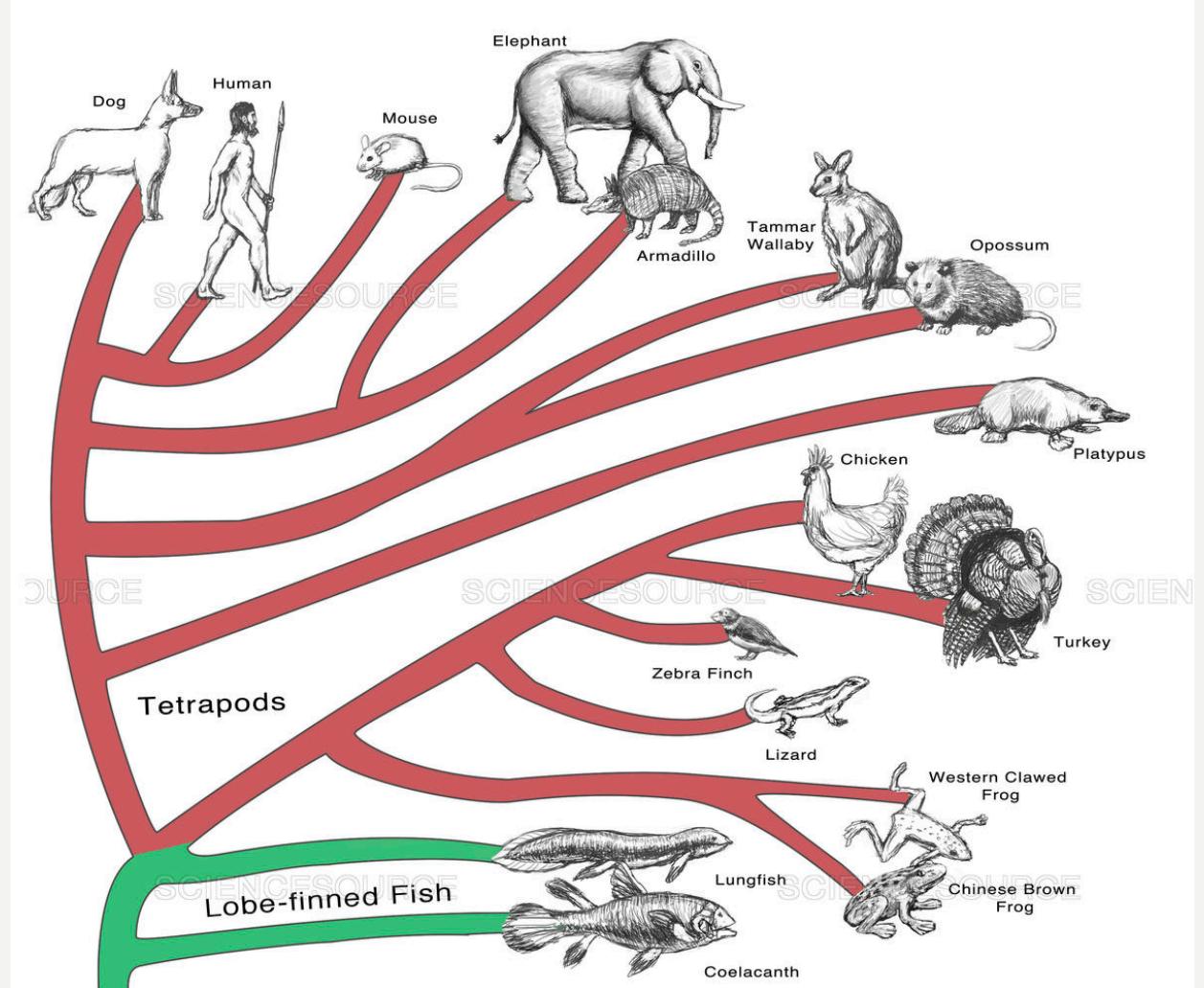
APPLICATIONS

- Many real-world relationships can be modeled as tree



a pedigree tree

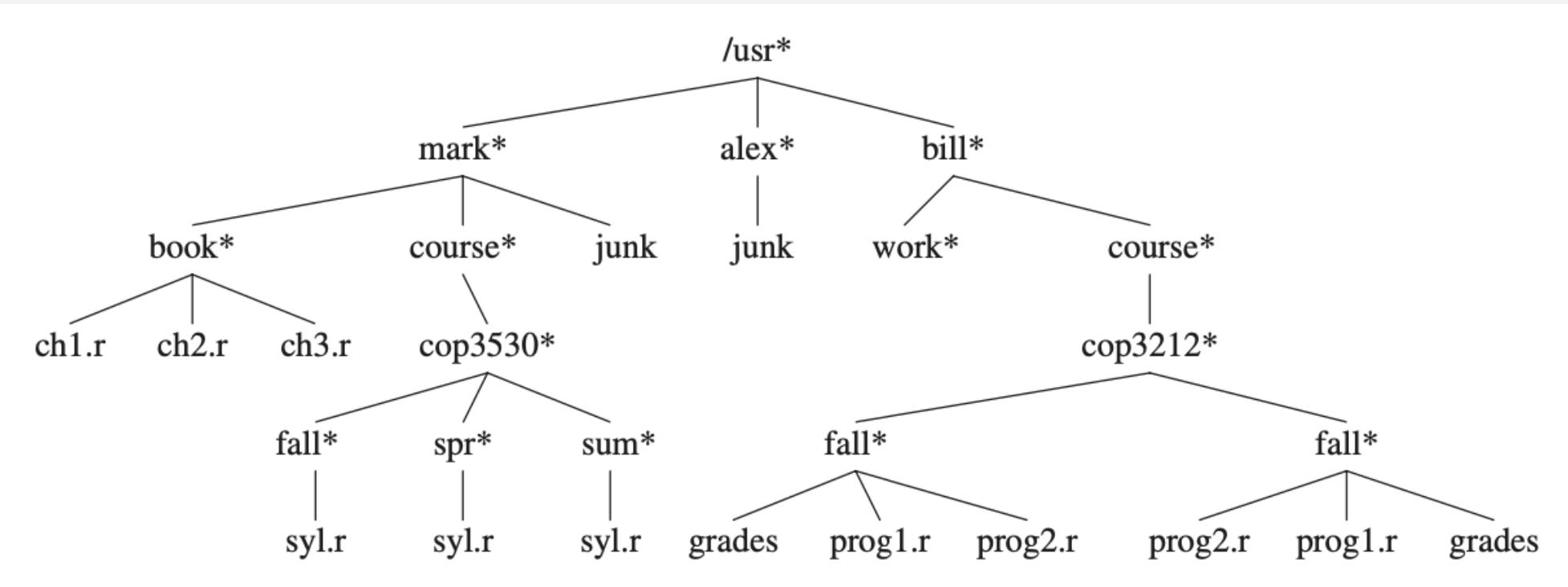
<https://www.ilovefreesoftware.com/01/featured/free-online-pedigree-chart-maker-websites.html>



a phylogenetic tree

<https://www.sciencesource.com/archive/Evolutionary-Tree--Animals-SS2737563.html>

APPLICATIONS



A Unix file system organization

APPLICATIONS

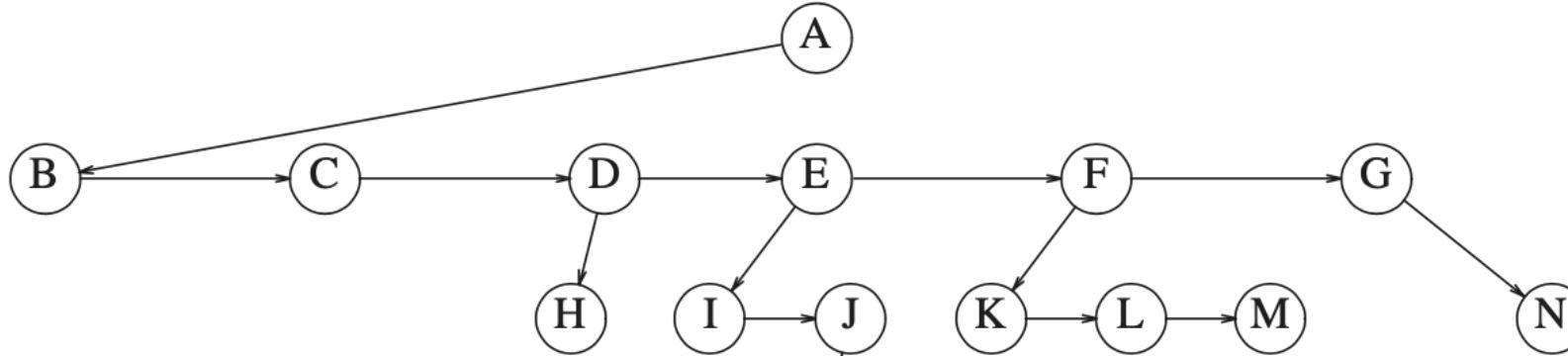
- More importantly, the tree ADT can be used as a data holder that balances the search performance between linked list and hash table
 - for linked list, searching for an element needs to go through the entire linked list and has a worst-case time complexity of $O(n)$
 - for hash table, searching for an element can be done in $O(1)$; however, it is associated with a large constant because the computation of the hash function is much slower than simply following pointers (as in the linked list ADT)
 - the tree ADT offers an alternative performance: a worst-case time complexity of $O(\log(n))$ with a small constant (only following links plus some simple comparisons)

IMPLEMENTATION

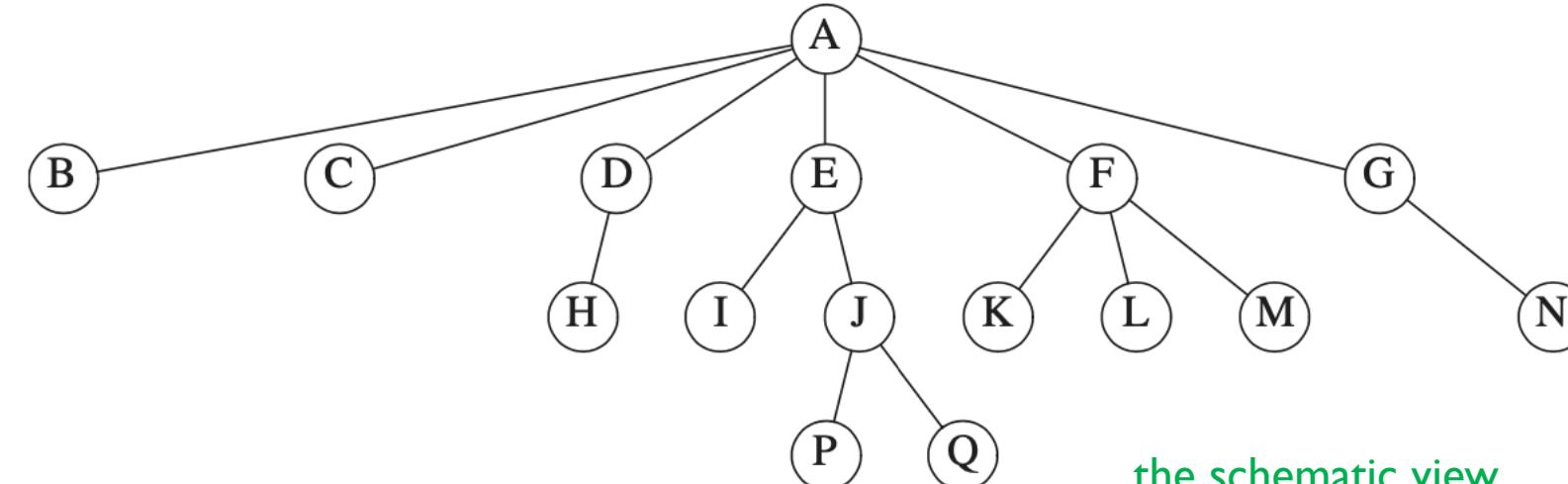
```
1 struct TreeNode
2 {
3     Object    element;
4     TreeNode *firstChild; // pointer that points to its leftmost child
5     TreeNode *nextSibling; // pointer that points to its next sibling
6 };
```

// we could add another pointer to the parent

IMPLEMENTATION



the implementational view



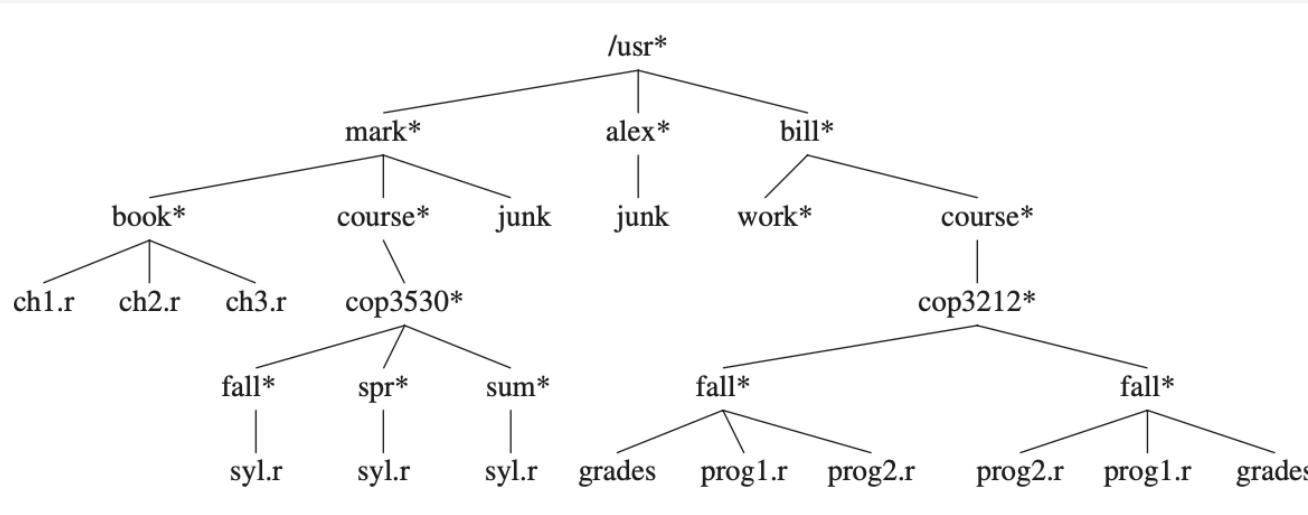
the schematic view

TREE TRAVERSAL

- Tree traversal: output all elements stored in the tree
- Tree traversal can be done in multiple ways (defined recursively)
 - preorder (current, left child,..., right child)
 - the current node is visited before the children, that's why it is named “pre”
 - postorder (left child,..., right child, current)
 - the current node is visited after the children, that's why it is called “post”

TREE TRAVERSAL

- Preorder application: listing the file directory path

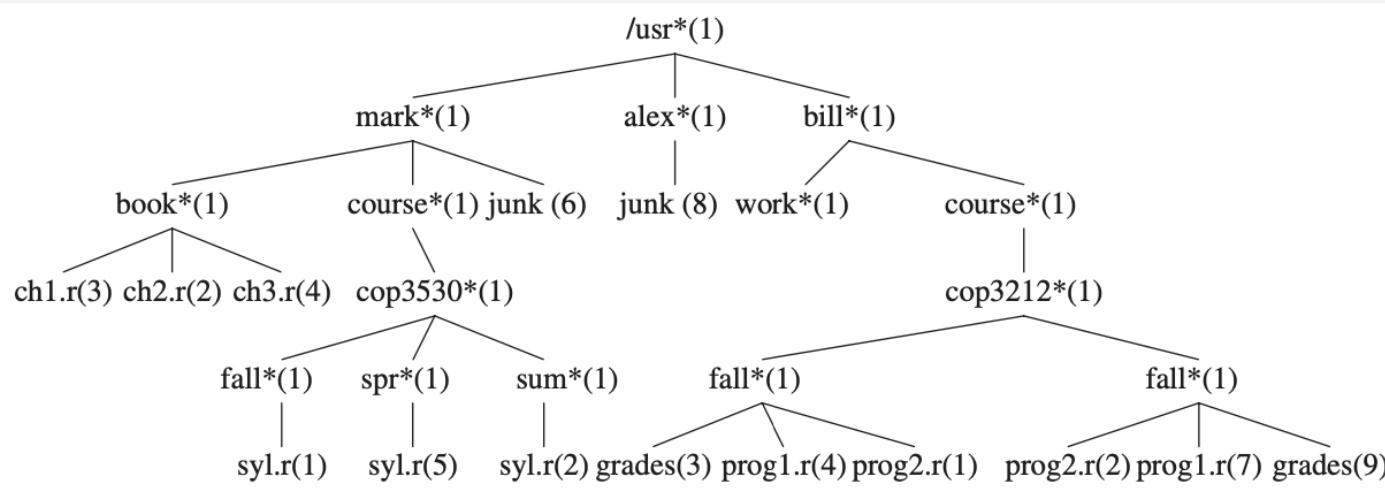


```
void FileSystem::listAll( int depth = 0 ) const
{
    printName( depth ); // Print the name of the object
    if( isDirectory( ) )
        for each file c in this directory (for each child)
            c.listAll( depth + 1 );
}
```

```
/usr
mark
book
ch1.r
ch2.r
ch3.r
course
cop3530
fall
syl.r
spr
syl.r
sum
syl.r
junk
alex
junk
bill
work
course
cop3212
fall
grades
prog1.r
prog2.r
fall
prog2.r
prog1.r
grades
```

TREE TRAVERSAL

- Postorder application: calculating the disk usage under each directory



```
int FileSystem::size( ) const
{
    int totalSize = sizeOfFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

    return totalSize;
}
```

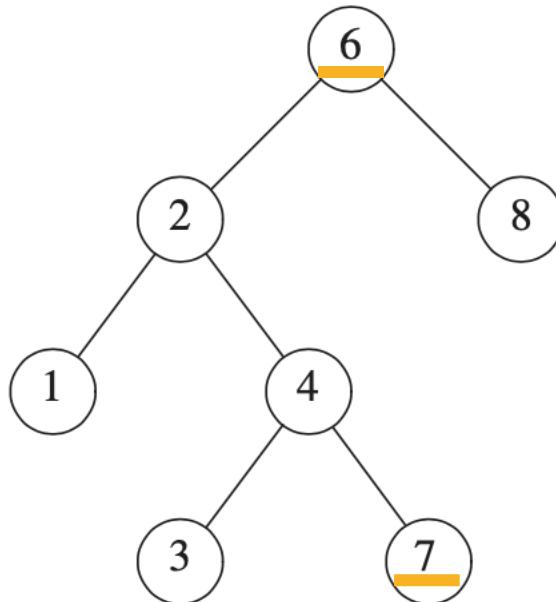
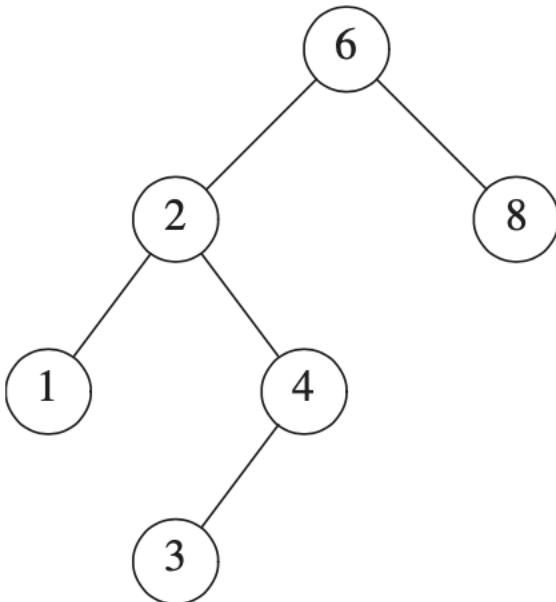
ch1.r	3
ch2.r	2
ch3.r	4
book	10
syl.r	1
fall	2
syl.r	5
spr	6
syl.r	2
sum	3
cop3530	12
course	13
junk	6
mark	30
junk	8
alex	9
work	1
grades	3
prog1.r	4
prog2.r	1
fall	9
prog2.r	2
prog1.r	7
grades	9
fall	19
cop3212	29
course	30
bill	32
/usr	72

BINARY TREE

- A binary tree is a tree that each of its node can contain no more than two children
- An important application of the binary tree is to facilitate information search
 - suppose we have a list of ordered element
 - for each element, we will ensure that all elements that are smaller than it must be placed in its left subtree, and all elements that are larger than it must be placed in its right subtree (this property defines a **binary search tree**)
 - when we perform the search, we will start from the root, and recursively compare the search key with the current element; if the key is smaller, we will go deeper into the left subtree and vice versa

BINARY TREE

- The left tree is a binary search tree, while the right one is not. Note that in the right tree, 7 is larger than 6, but it is placed in the left subtree of 6.



BINARY SEARCH TREE: OPERATIONS

- The binary search tree (BST) ADT has three fundamental operations:
 - findMin: returns the smallest element in the BST
 - finxMax: returns the largest element in the BST
 - contains: searches against the BST to determine whether an element exists
 - insert: insert a new element into the BST
 - remove: remove an existing element from the BST

BINARY SEARCH TREE: OPERATIONS

- definition

```
struct BinaryNode
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
        : element{ theElement }, left{ lt }, right{ rt } { }

    BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
        : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }

};
```

BINARY SEARCH TREE: OPERATIONS

- `findMin()` and `findMax()`: recursively find the leftmost or rightmost leaf node

```
1  /**
2   * Internal method to find the smallest item in a subtree t.
3   * Return node containing the smallest item.
4   */
5  BinaryNode * findMin( BinaryNode *t ) const
6  {
7      if( t == nullptr )
8          return nullptr;
9      if( t->left == nullptr )
10         return t;
11      return findMin( t->left );
12 }
```

```
1  /**
2   * Internal method to find the largest item in a subtree t.
3   * Return node containing the largest item.
4   */
5  BinaryNode * findMax( BinaryNode *t ) const
6  {
7      if( t != nullptr )
8          while( t->right != nullptr )
9              t = t->right;
10     return t;
11 }
```

BINARY SEARCH TREE: OPERATIONS

- contains(): this is essentially a binary search

```
1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6 bool contains( const Comparable & x, BinaryNode *t ) const
7 {
8     if( t == nullptr )
9         return false;
10    else if( x < t->element )
11        return contains( x, t->left );    // a recursive call
12    else if( t->element < x )
13        return contains( x, t->right );  // a recursive call
14    else
15        return true;      // Match
16 }
```

BINARY SEARCH TREE: OPERATIONS

- `insert()`: find the correct location to insert in a way similar to `contains()`

```
1  /**
2   * Internal method to insert into a subtree.
3   * x is the item to insert.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6  */
7 void insert( const Comparable & x, BinaryNode * & t )
8 {
9     if( t == nullptr )
10        t = new BinaryNode{ x, nullptr, nullptr };
11    else if( x < t->element )
12        insert( x, t->left );      // a recursive call
13    else if( t->element < x )
14        insert( x, t->right );    // a recursive call
15    else
16        ; // Duplicate; do nothing
17 }
```

BINARY SEARCH TREE: OPERATIONS

- `remove()`:

```
1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6  */
7 void remove( const Comparable & x, BinaryNode * & t )
8 {
9     if( t == nullptr )
10        return; // Item not found; do nothing
11    if( x < t->element )
12        remove( x, t->left );      // a recursive call
13    else if( t->element < x )
14        remove( x, t->right );    // a recursive call
15    else if( t->left != nullptr && t->right != nullptr ) // Two children
16    {
17        t->element = findMin( t->right )->element;
18        remove( t->element, t->right );
19    }
20    else
21    {
22        BinaryNode *oldNode = t;
23        t = ( t->left != nullptr ) ? t->left : t->right;
24        delete oldNode;
25    }
26 }
```

// if the node to be removed has two children, use the smallest element in the right subtree to replace the current node

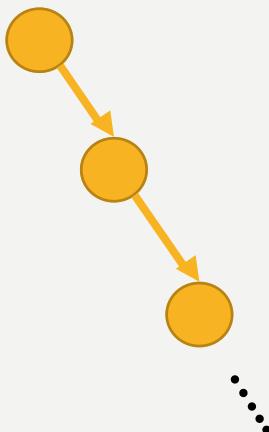
// if the node to be removed has only one child, directly link the node's parent to its child

BINARY SEARCH TREE: OPERATIONS

- Expected time complexity: how many steps do we expect to take before hitting the correct location of the tree (the traveled path length)?
 - if we pick a random element as the root, we can expect that half of the elements are going to be smaller than it and half of the elements are going to be larger than it. In this case, we can expect that the left subtree has the same size as the right subtree
 - we can make this assumption recursively for each internal node, so we get a tree that is as “wide” as possible, where the number of nodes in each layer doubles as we go deeper
 - in this case, we can expect of tree depth of $O(\log(n))$
 - more formal analysis will be introduced later in the class

BINARY SEARCH TREE: THE WORST-CASE SCENARIO

- The tree can be very deep and degenerates to a linked list if we are adding in sorted elements
 - e.g., the second element is larger than the first element, so we add the second element as the right child of the first element
 - the third element is larger than the second element, so we add the third element as the right child of the second element
 -



BINARY SEARCH TREE: THE WORST-CASE SCENARIO

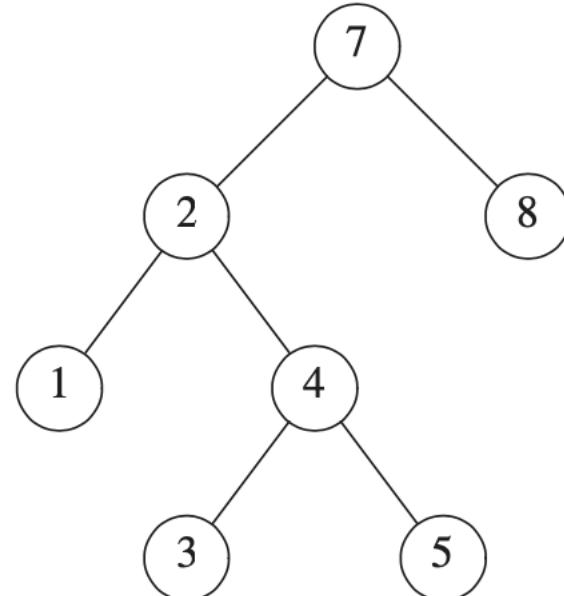
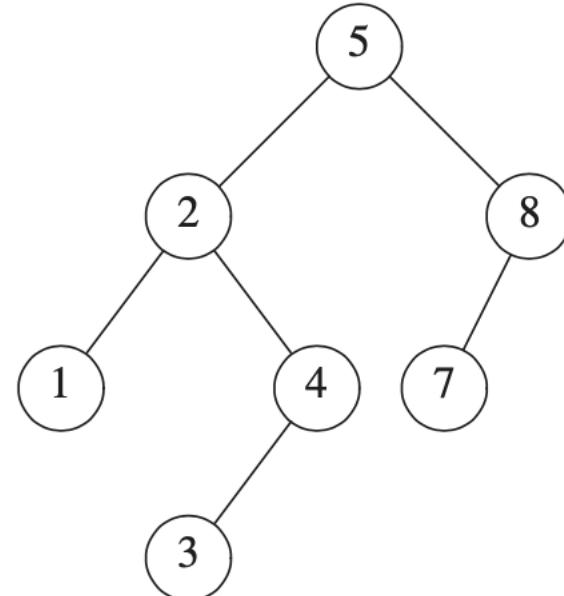
- In this case, the expected $O(\log(n))$ search/insert/remove time becomes $O(n)$
- To avoid the worst-case scenario, we can perform additional operations (during tree construction) to alter the tree topology, and make the tree as “wide” and as “flat” as possible (we call such a tree balanced)

BINARY SEARCH TREE: THE AVL TREE

- AVL is due to Adelson, Velskii, and Landis
- They proposed a balance property for binary search tree (and ways to maintain this property)
 - the AVL property: for each node, the height of its left subtree and the height of its right subtree cannot differ by more than 1
- A binary search tree that satisfies the AVL property is called an AVL tree
- An AVL tree guarantees $O(\log(n))$ search/insert/remove time

BINARY SEARCH TREE: THE AVL TREE

- The left tree is an AVL tree, the right one is not.
 - consider node 7 (the root), its left tree depth is 3, while its right tree depth is 1



BINARY SEARCH TREE: THE AVL TREE

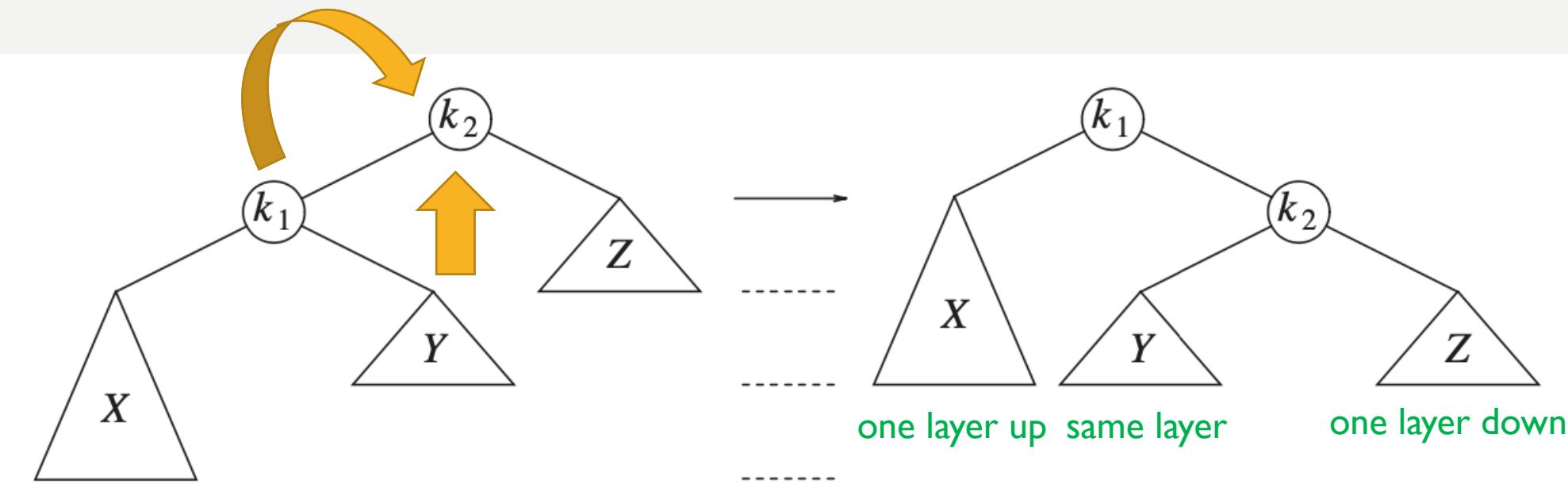
- We will need to maintain the AVL property when we alter the topology of the tree
 - insertion and deletion will modify the topology
 - we will use insertion cases as examples to illustrate how to maintain the AVL property
- Given a node α , There are 4 insertion cases we need to consider:
 - we are inserting to the left subtree of the left child of α
 - we are inserting to the right subtree of the left child of α
 - we are inserting to the left subtree of the right child of α
 - we are inserting to the right subtree of the right child of α
 - the first and fourth cases are essentially the same (symmetric), and the second and the third cases are essentially the same as well

BINARY SEARCH TREE: THE AVL TREE

- For the first and fourth cases, we will use a simpler “single rotation” approach
- For the second and third cases, we will use a slightly more sophisticated “double rotation” approach

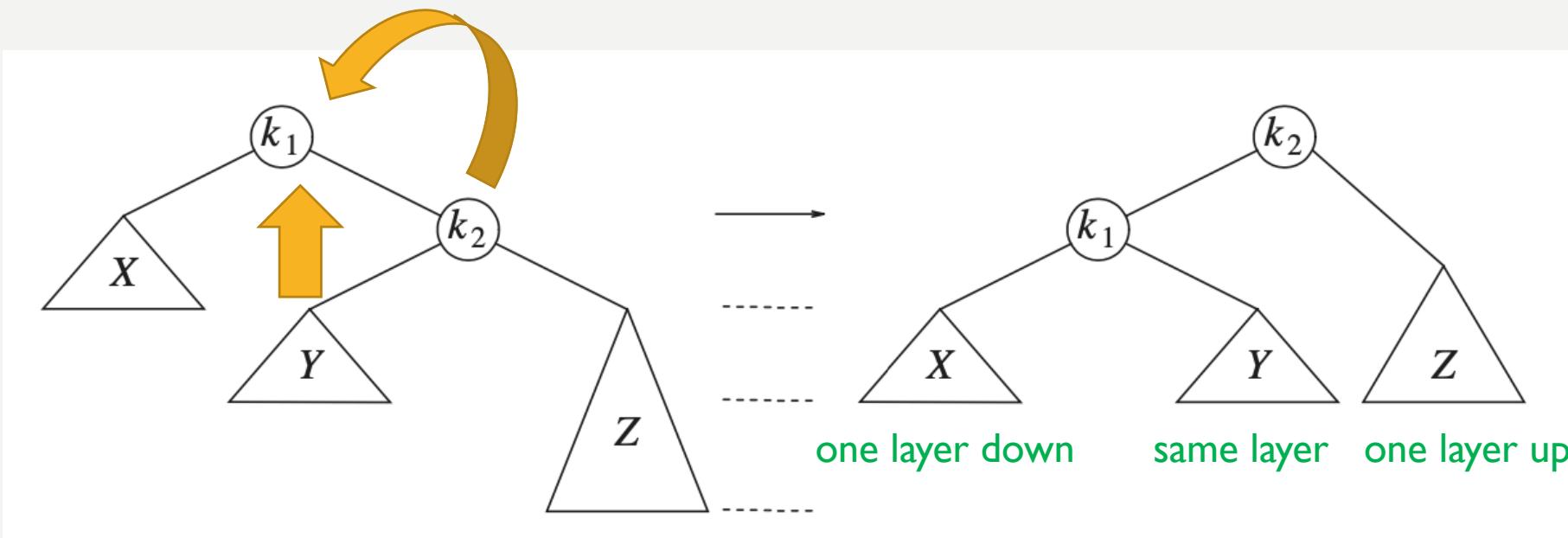
AVL TREE: SINGLE ROTATION

- Consider case I, where we are inserting an element into the subtree X; it increases the height of X by 1 and results in the violation of the AVL property at the node k2
- We will rotate k1 and k2 clockwise, and detach the subtree Y (if non-empty) from k1 and reattach it to k2



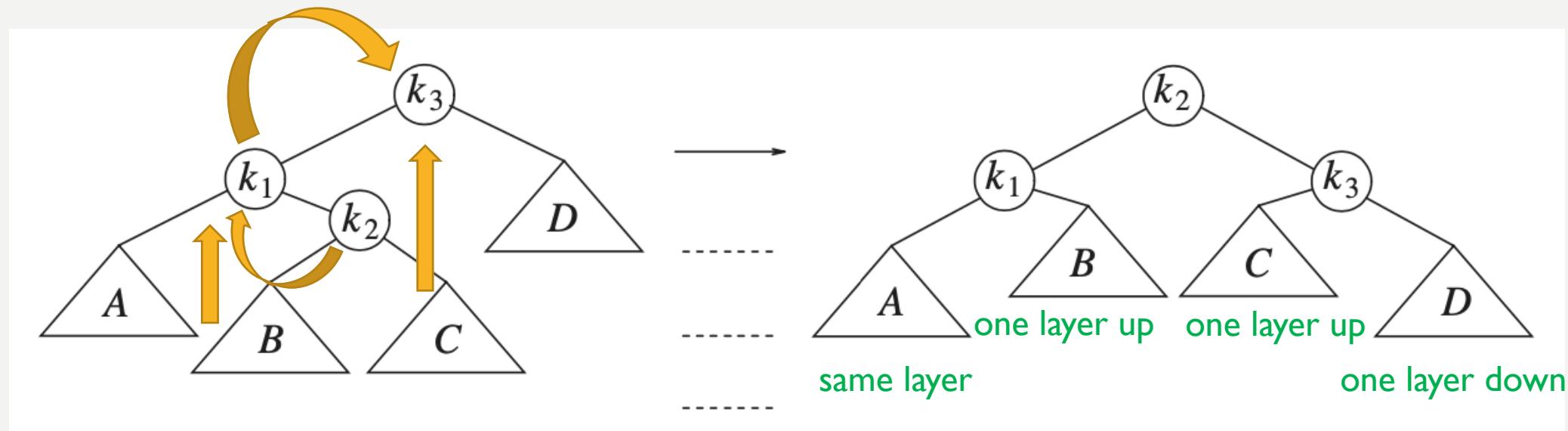
AVL TREE: SINGLE ROTATION

- Consider case 4, where we are inserting an element into the subtree Z; it increases the height of Z by 1 and results in the violation of the AVL property at the node k1
- We will rotate k1 and k2 anti-clockwise, and detach the subtree Y (if non-empty) from k2 and reattach it to k1



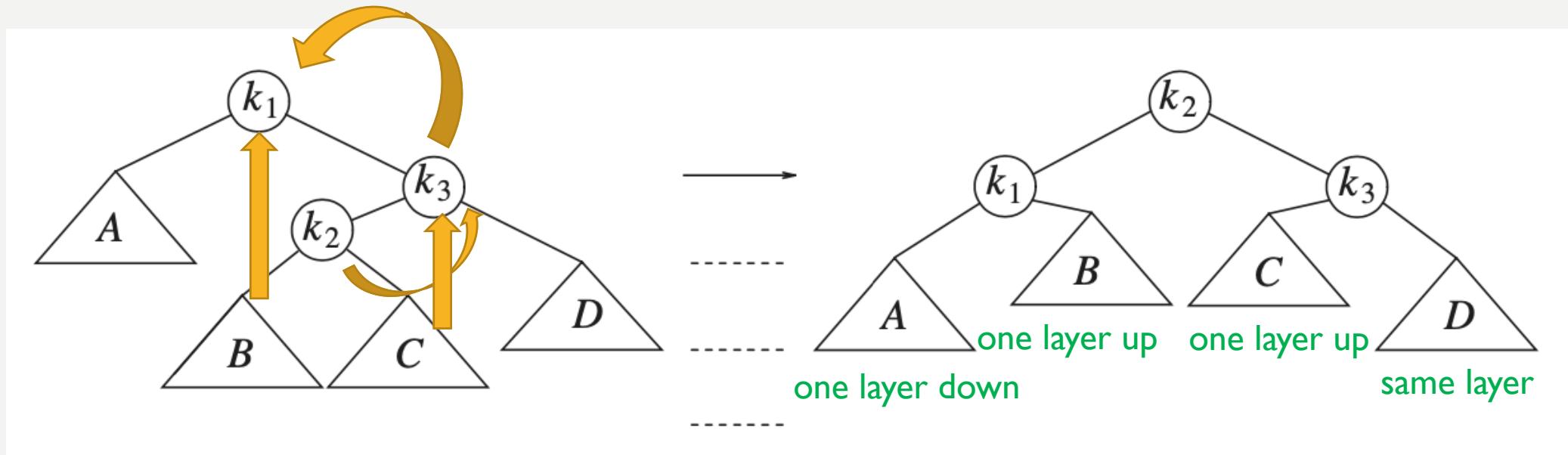
AVL TREE: DOUBLE ROTATION

- Consider case 2, where we are inserting an element into the subtree B or C; it increases either of their heights by 1 and results in the violation of the AVL property at the node k3
- We will rotate k1 and k2 clockwise, and then rotate k2 and k3 clockwise (hence double rotation). We will also detach the subtrees B and C (if non-empty) from k2, reattach B to k1 and C to k3.



AVL TREE: DOUBLE ROTATION

- Consider case 3, where we are inserting an element into the subtree B or C; it increases either of their heights by 1 and results in the violation of the AVL property at the node k_1
- We will rotate k_1 and k_2 clockwise, and then rotate k_2 and k_3 clockwise (hence double rotation). We will also detach the subtrees B and C (if non-empty) from k_2 , reattach B to k_1 and C to k_3 .



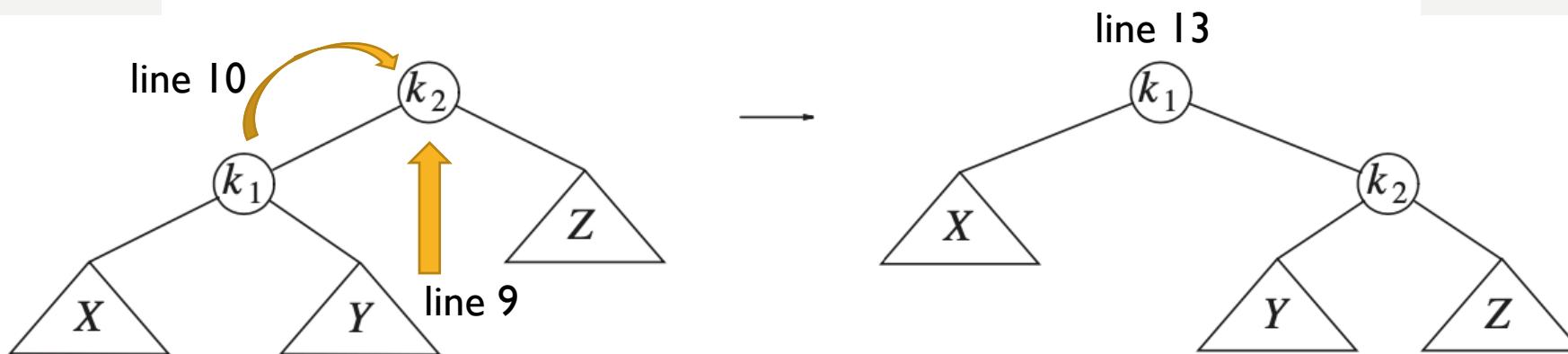
AVL TREE: IMPLEMENTATION

- We will be recording the height of each node for AVL tree, while the other components stay the same

```
1 struct AvlNode
2 {
3     Comparable element;
4     AvlNode *left;
5     AvlNode *right;
6     int      height; // now adding subtree height
7
8     AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
9         : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10
11    AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12        : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13};
```

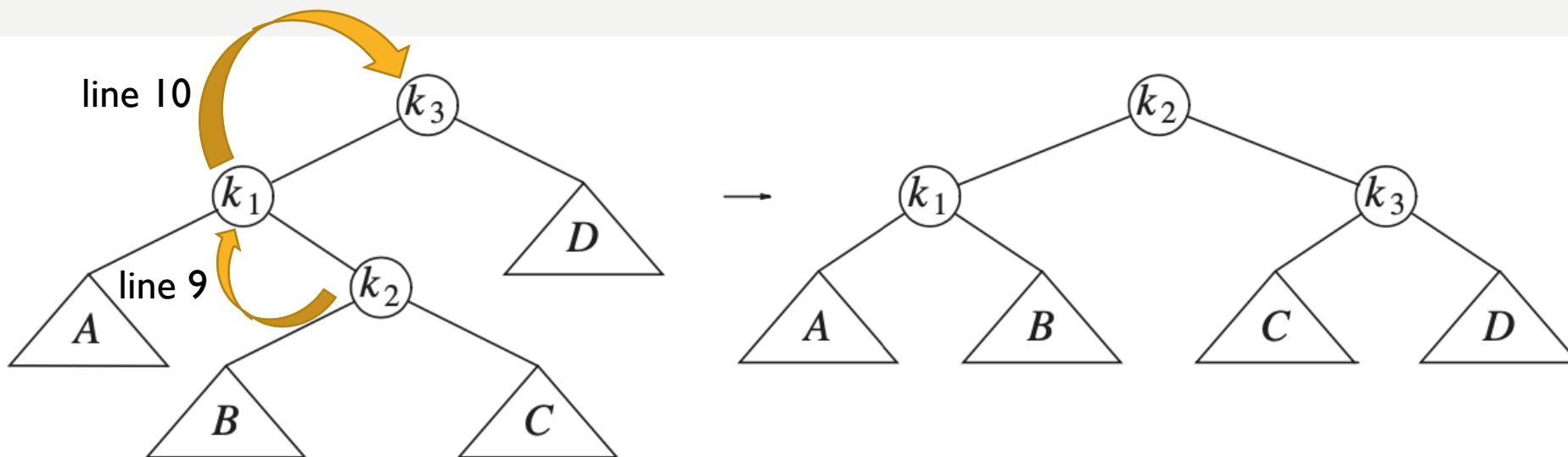
AVL TREE: IMPLEMENTATION

```
1  /**
2   * Rotate binary tree node with left child.
3   * For AVL trees, this is a single rotation for case 1.
4   * Update heights, then set new root.
5   */
6 void rotateWithLeftChild( AvlNode * & k2 )
7 {
8     AvlNode *k1 = k2->left;
9     k2->left = k1->right;
10    k1->right = k2;
11    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12    k1->height = max( height( k1->left ), k2->height ) + 1;
13    k2 = k1;
14 }
```



AVL TREE: IMPLEMENTATION

```
7 void doubleWithLeftChild( AvlNode * & k3 )
8 {
9     rotateWithRightChild( k3->left );
10    rotateWithLeftChild( k3 );
11 }
```



AVL TREE: IMPLEMENTATION

- balance the subtree rooted at t

```
21 // Assume t is balanced or within one of being balanced
22 void balance( AvlNode * & t )
23 {
24     if( t == nullptr )
25         return;
26
27     if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28         if( height( t->left->left ) >= height( t->left->right ) )
29             rotateWithLeftChild( t );
30         else
31             doubleWithLeftChild( t );
32     else
33         if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34             if( height( t->right->right ) >= height( t->right->left ) )
35                 rotateWithRightChild( t );
36             else
37                 doubleWithRightChild( t );
38
39     t->height = max( height( t->left ), height( t->right ) ) + 1;
40 }
```

// if the AVL property is violated
// case 1

// case 2

// if the AVL property is violated
// case 4

// case 3

AVL TREE: IMPLEMENTATION

```
7 void insert( const Comparable & x, AvlNode * & t )
8 {
9     if( t == nullptr )                                // locate where to insert the node
10        t = new AvlNode{ x, nullptr, nullptr };
11    else if( x < t->element )
12        insert( x, t->left );
13    else if( t->element < x )
14        insert( x, t->right );
15
16    balance( t );                                    // re-balance the tree if necessary
17 }
```

AVL TREE: IMPLEMENTATION

```
7 void remove( const Comparable & x, AvlNode * & t )      // deleting a node could make a subtree shorter, which is
8 {                                                     similar to make its sibling higher; so we can apply
9     if( t == nullptr )                                rebalancing in a similar way if necessary
10    return;   // Item not found; do nothing
11
12    if( x < t->element )
13        remove( x, t->left );
14    else if( t->element < x )
15        remove( x, t->right );
16    else if( t->left != nullptr && t->right != nullptr ) // Two children
17    {
18        t->element = findMin( t->right )->element;
19        remove( t->element, t->right );
20    }
21    else
22    {
23        AvlNode *oldNode = t;
24        t = ( t->left != nullptr ) ? t->left : t->right;
25        delete oldNode;
26    }
27
28    balance( t );      // add rebalancing
29 }
```

B-TREE: MOTIVATION

- Now we have established that the advantage of using the tree ADT for information storage and retrieval will allow $O(\log(n))$ time
 - and we can achieve this purely through following the edges, without spending time to compute the hash function as in the hash table ADT
- More precisely, if we are using a BST, the search/insert/delete time complexity is $O(\log_2(n))$

B-TREE: MOTIVATION

- In practice, information in the tree is not stored in the main memory, it is stored in the hard disk (because we do not have that much memory to hold all information).
- Accessing hard disk is much slower than accessing the main memory. To facilitate faster access, related information is often put into adjacent disk blocks and are loaded into the main memory altogether in a single access (called prefetch). The underlying rationale is assuming related information is often accessed altogether.
- For the tree ADT, data stored in the same node can be considered as related information and can be retrieved in a single hard disk access.

B-TREE: MOTIVATION

- As a result, each time we follow an edge to locate the tree element for search insertion/deletion, we will incur a hard disk access. The expected number of hard disk access is $O(\log_2(n))$.
- An intuitive way to reduce the number of hard disk access is to increase the number of children each node can have. For example, if we allow M children per node, we can expect the number of hard disk access being reduced to $O(\log_M(n))$.
- Indeed, finding the correct edge to follow among M edges is more challenging than finding the correct one between 2 edges. However, we are willing to pay the extra computation as the benefit we get from reducing hard disk access is worthy.

B-TREE: DEFINITION

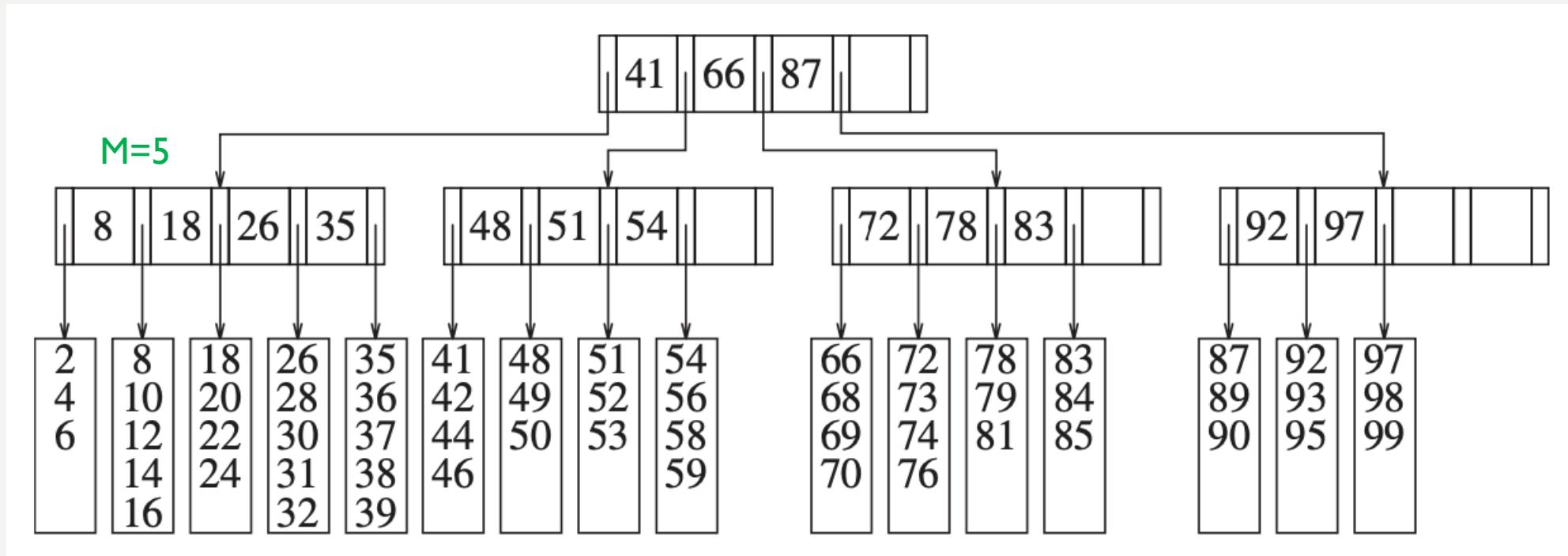
- Such a M-ary search tree allows M-way branching. We ensure the following properties to make it reasonably balanced, which ensures the expected $O(\log M(n))$ search/insert/delete time.
- The properties we will ensure:
 - the data items are stored in leaves
 - the non-leaf nodes store up to $M-1$ keys to guide the searching; key i represents the smallest key in subtree $i + 1$
 - the root is either a leaf or has between 2 and M children
 - all non-leaf nodes (except the root) have between $\text{ceiling}(M/2)$ to M children
 - all leaf nodes are at the same depth and have between $\text{ceiling}(L/2)$ and L data items
- An M-ary search tree that satisfies the above properties is called a **B-tree**.

B-TREE: DEFINITION

- While examine the B-tree properties, we found:
 - the tree is only a bit worse than half-full (by requiring all internal nodes to have at least $\text{ceiling}(M/2)$ children and all leaf nodes to have at least $\text{ceiling}(L/2)$ data items)
 - the tree is balanced (by requiring all leaf nodes to have the same depth)
- We can therefore expect the $O(\log M(n))$ search/insert/delete time.

B-TREE: DEFINITION

- A B-tree example with $M=5$ and $L=5$:



$L=5$

B-TREE: OPERATIONS

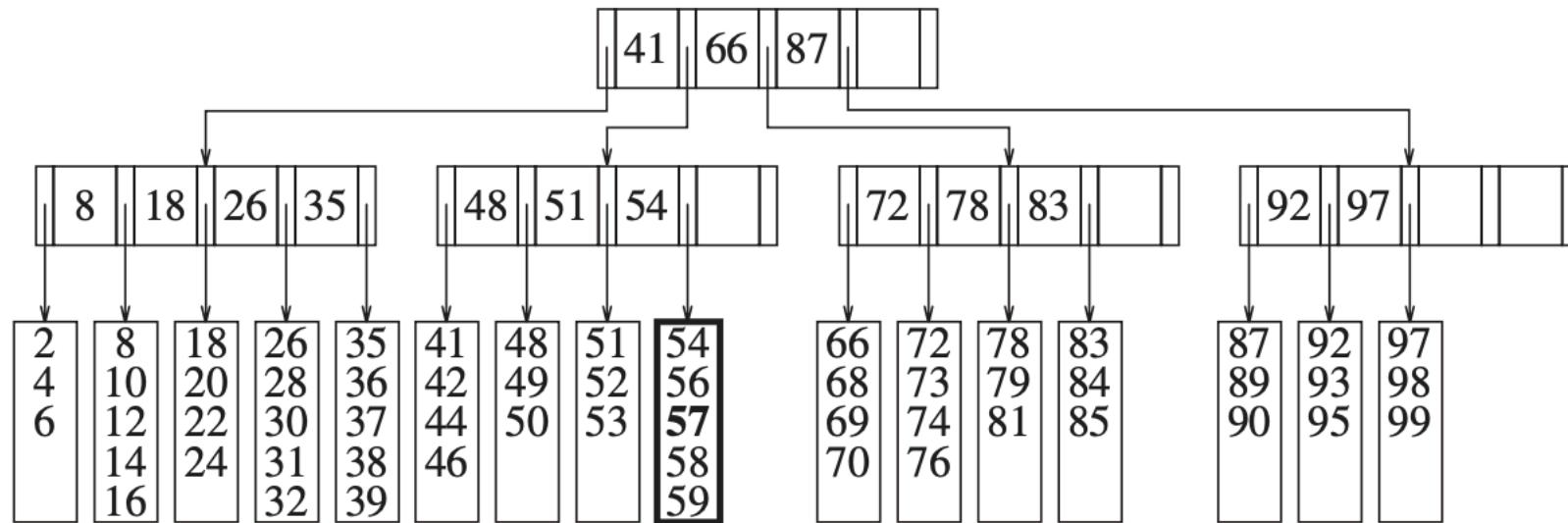
- Search
 - traverse down the B-tree in a similar way as the BST traversal
 - except that we need to compare they search key with each of the M values to determine the correct edge to follow
 - this is OK because all of the M values are stored in an adjacent location of the hard disk and can be retrieved with only one access
 - once we get to the leaf node, compare the key with each data item to complete the search

B-TREE: OPERATIONS

- insert
 - use the search procedure to find the data block to insert
 - however the insertion may change the tree topology, and we need to maintain all B-tree properties
 - case 1: no split needed
 - case 2: one-level split
 - case 3: recursive split

B-TREE: OPERATIONS

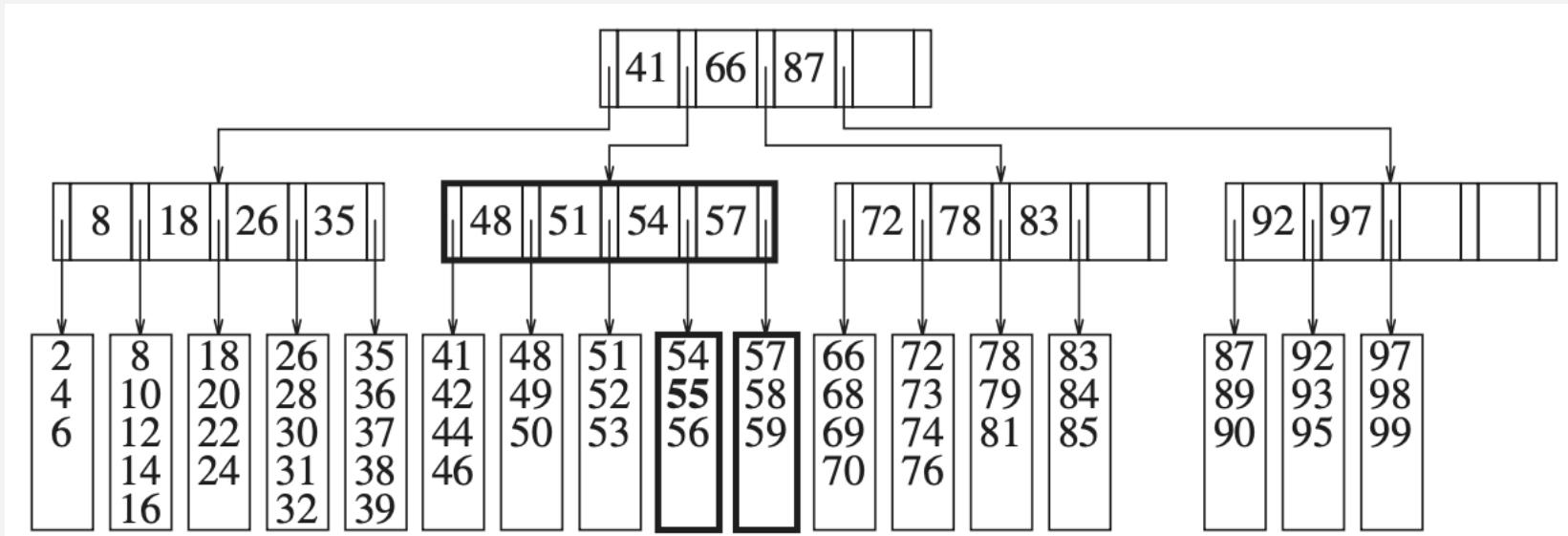
- insertion: if the current data block still has space, no node split is needed



insertion of 57

B-TREE: OPERATIONS

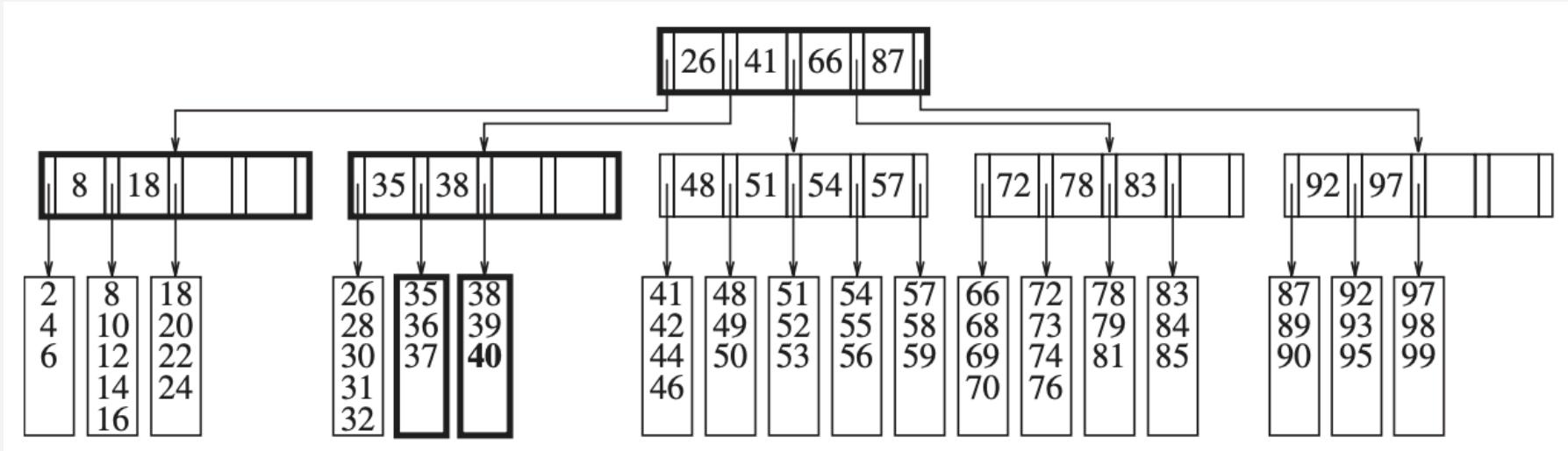
- insertion: if the current data block is full, we will split the data block and add a new child to its parent



- 1: Insertion of 55, the block becomes full.
- 2: The block is split into 2, the new block is attached to the parent.
- 3: The two split blocks each contains ceiling($L/2$) data items.
- 4: The parent contains no more than M items.

B-TREE: OPERATIONS

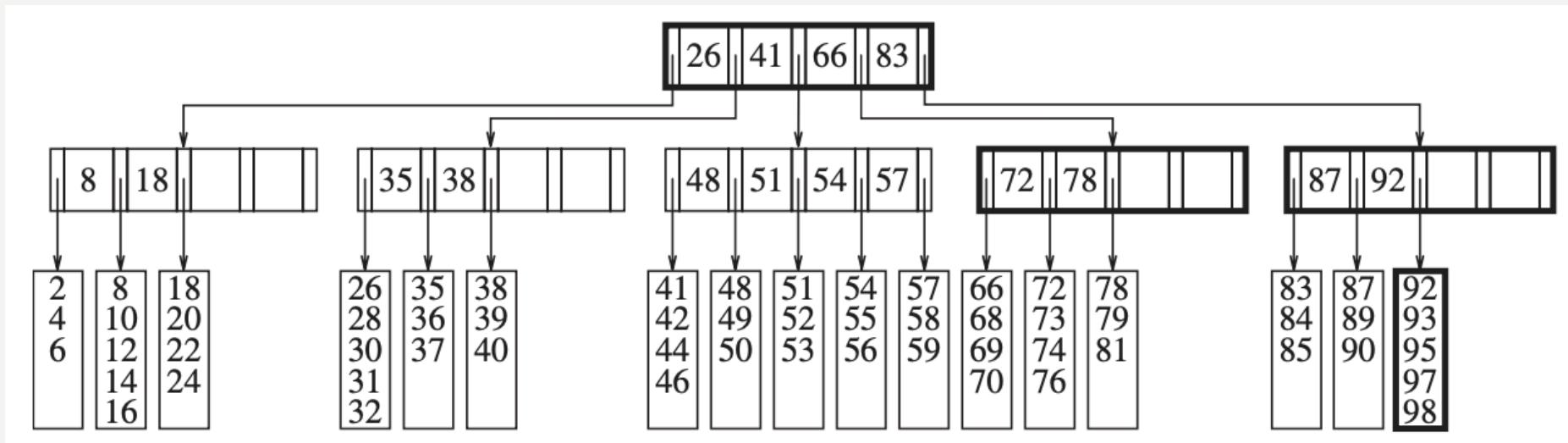
- insertion: if the parent is also full, we will split parent block and do it recursively



- 1: Insertion of 40, the block becomes full.
- 2: The block is split into 2, the new block is attached to the parent.
- 3: The parent is also full, the parent is split into 2.
- 4: The newly-split parents are further attached to the grandparent.
- 5: If we need to split the root, we will create a new root as the parent of the two split root. (Note that the root can contain as few as 2 children.)

B-TREE: OPERATIONS

- deletion: the idea is similar to insertion; we will perform (recursive) merge instead of split if necessary



- I: Deletion of 99, the block contains less than ceiling(L/2) items.
- 2: Merge it with its previous data block.
- 3: The parent contains less than ceiling(M/2) data items.
- 4: Borrow a data block from its sibling.

C++ STL

- std::map and std::set implement the search tree ADT.
- std::map stores key-value pairs (similar to std::unordered_map)
 - faster traversal, slower search/insert/delete in general as compared to std::unordered_map
- std::set stores single values (similar to std::unordered_set)
 - faster traversal, slower search/insert/delete in general as compared to std::unordered_set

SUMMARY

- The tree ADT is an important data structure for modeling many natural relationships.
- It further supports $O(\log(n))$ time search/insert/delete with a small associated constant.
 - linked list requires $O(n)$ time with a small constant
 - hash table requires $O(1)$ time with a large constant
- The property is ensured if the tree is balanced
 - AVL trees: binary search tree that satisfies the AVL property. We use rotation to ensure the AVL property in cases of insertion and deletion.
 - B-tree: hard disk-friendly data structure. Essentially an M-ary search tree. We use split and merge to ensure the B-tree property in cases of insertion and deletion.
 - Splay tree (not discussed in-class but available in the textbook): bring elements close to the root such that the subsequent access of them can be made faster (like bubbling up the nodes up the tree). Study if you are interested; will not test it in the exams.

EECS 560

DATA STRUCTURES

MODULE V: PRIORITY QUEUE

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++, 4th edition*, by Mark Allen Weiss.

PRIORITY QUEUE AND ITS APPLICATIONS

- Priority queue is an extension of queue. Unlike the queue ADT which has a first-in-first-out property, the priority queue determines the “first-out” element using a user-defined property that may be different than the enqueue order.
- For example, imagine we are running a server. Rather than simply executing the job that is submitted at the earliest time, we want to execute the job that pays us the most. Here, the property “pays us the most” is defined by us to determine the execution order of the jobs.

PRIORITY QUEUE: FUNDAMENTAL OPERATIONS

- Similar to the queue ADT, the priority queue ADT also has two fundamental operations:
 - enqueue(): insert an element into the queue
 - findHighestPriority() and deleteHighestPriority(): find and delete the element that has the highest priority. Note that these functions are similar to the dequeue() function in the queue ADT. (And in the book they are referred as findMin() and deleteMin())

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using array?
- Can we implement this ADT using linked list?
- Can we implement this ADT using hash table?
- Can we implement this ADT using binary search tree?

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using array?
 - **insert without sorting:** We can directly insert the element to the last position of the array, which will take $O(1)$ time. If the array is not sorted, then we need to go through the entire array to find the element with the highest priority, which will take $O(n)$ time.
 - **insert with sorting:** If the elements in the array are sorted, then we can spend $O(\log(n))$ time to find the correct position for the new element, and then spend $O(n)$ time to move the existing elements to spare a space for the new element. When finding the element with the highest priority, we can simply take the first element using only $O(1)$ time. (Note that if we need to delete the element, we will need an extra $O(n)$ time to move the remaining elements.)

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using linked list?
 - **insert without sorting:** This is the same as if we implement using array. That is, $O(1)$ time for insert and $O(n)$ time for locating the element with the highest priority.
 - **insert with sorting:** We will simply go through the linked list to insert the element to the proper location, which will take $O(n)$ time. (Note that unlike array, linked-list does not allow binary search because we cannot directly access an element by its index in $O(1)$ time.) Accessing the element with the highest priority, including deleting it, will both take $O(1)$ time.

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using hash table?
 - Note that the hash function may not preserve priority. That is, the key with a higher priority does not necessarily mean it will be hashed into a lower (or a higher) position in the hash table. (If yes then it is essentially an array.)
 - In this case, elements in the hash table will not be sorted. Insertion will thus simply take $O(1)$ time; and the search/deletion will take $O(n)$ time.

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using binary search tree?
 - Yes, we will be able to do search/insertion/deletion in $O(\log(n))$ time. In this case, inserting an element and finding the element with the highest priority will both take $O(\log(n))$ time.

PRIORITY QUEUE: MOTIVATION

- Can we do better?
- For array, linked list, and hash table, all of them requires an $O(n)$ time for either an insertion or finding the element with the highest priority.
- For binary search tree, both insertion and search will take $O(\log(n))$ time. However, we observe that many operations are wasted on maintaining the BST properties. While it allows the search of any element in $O(\log(n))$ time, we only need to search for the one with the highest priority. So, can we make the BST simpler to allow faster time for searching the element with highest priority?

PRIORITY QUEUE: MOTIVATION

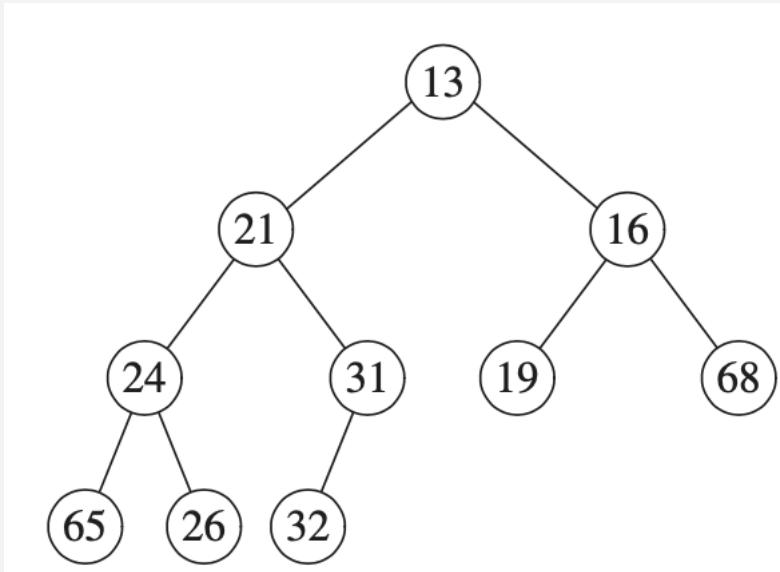
- Our target:
 - $O(\log(n))$ time for insertion
 - $O(1)$ time for finding the element with the highest priority
 - $O(\log(n))$ time for deletion
- Expected advantage over BST
 - $O(1)$ time access vs. $O(\log(n))$ time access of the element with the highest priority
 - will use pure array implementation (no pointer and no scattered data blocks in the hard disk), hence more computational and space efficient

PRIORITY QUEUE: HEAP IMPLEMENTATION

- In many cases, “priority queue” and “heap” are used interchangeably. I personally prefer to use “priority queue” to describe the property and behavior of the ADT, and use “heap” in the context of implementation.

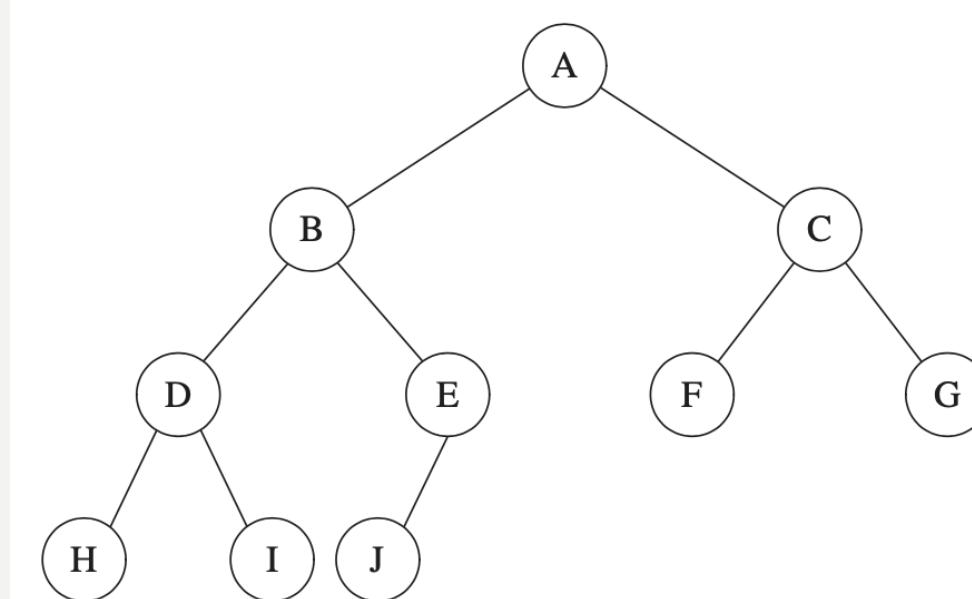
HEAP PROPERTY

- A heap looks like a binary search tree, however it has different properties:
 - a heap is full (elements attached from left to right, layer by layer) while a BST is only balanced (referred as structure property)
 - in heap, the parent is smaller than its children (and we don't care which child is larger); in BST, the parent is larger than its left child and smaller than its right child (referred as heap-order property)



HEAP IMPLEMENTATION

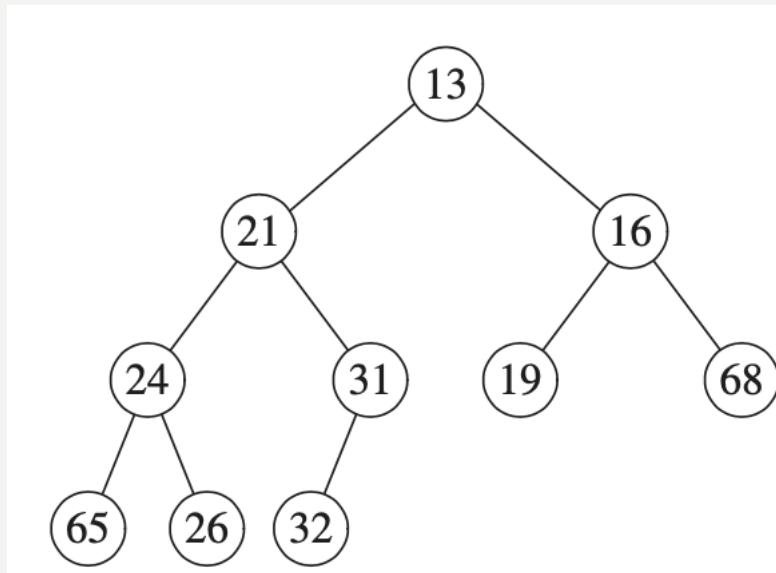
- Because the heap is full, we can use an array to store it (and get rid of the space-consuming pointers).
- For any element, we can always derive its index in the array from its position in the hypothetical binary tree



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

HEAP OPERATION: SEARCH

- Finding the element with the highest priority (consider the numbers in the examples as the “rank”, so the element with the highest priority correspond to the smallest number)
 - This is trivial with the heap property, which states that any parent is smaller than its children. In this case, the root will be the element with the highest priority (note that we do not delete it; will discuss deletion later)



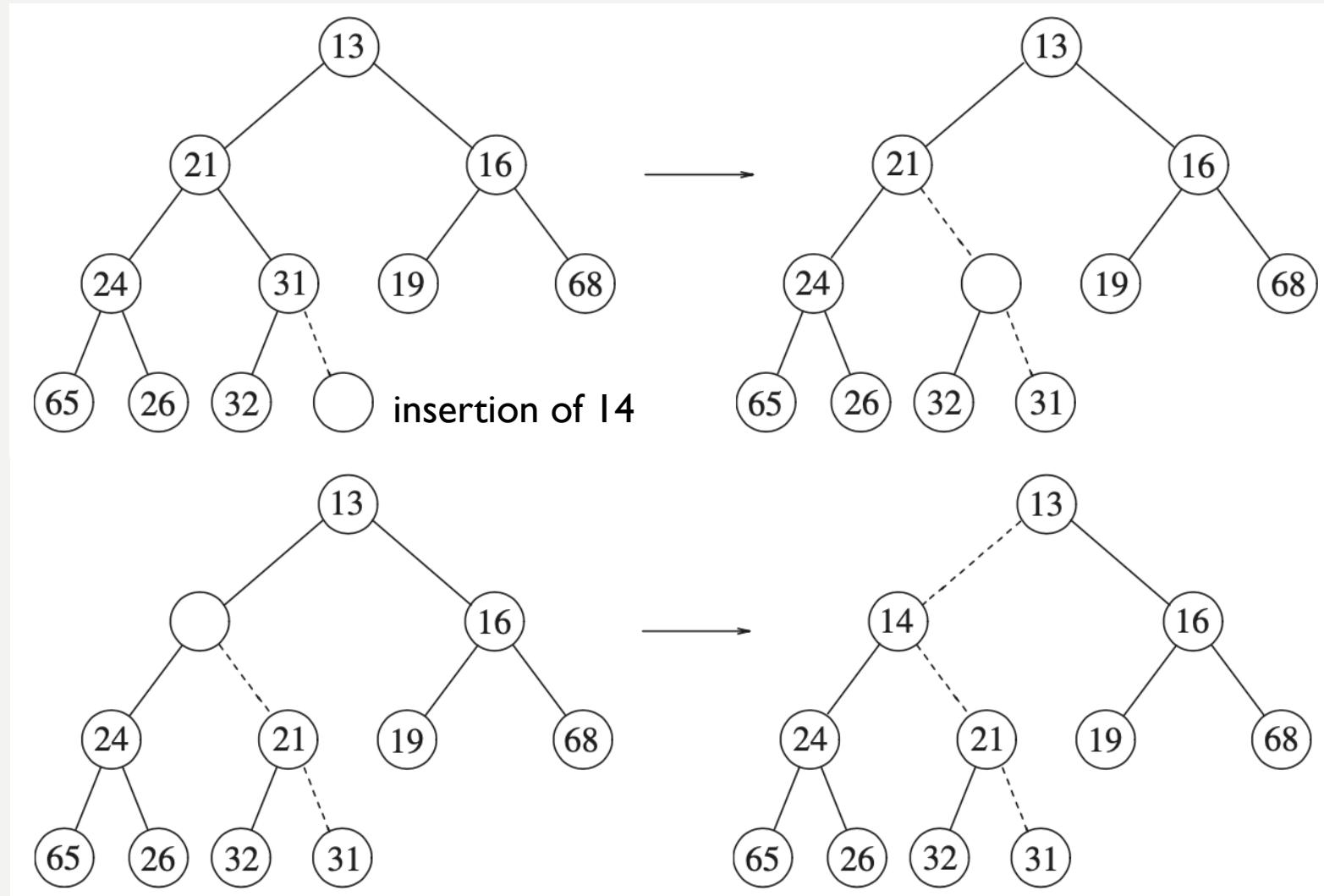
HEAP OPERATION: INSERT

- `insert()`: Note that the element we insert may have an arbitrary priority.
- Given the structure property of heap (that is, the binary tree is full), the element must be placed on the right-most position of the lowest layer of the tree.
- However, placing the new element on this position may violate the heap property (for example, it may be smaller than its parent). Therefore, we need to adjust the tree if necessary. The process is called “percolation”.

HEAP OPERATION: INSERT

- We know that if the newly-inserted element is larger than its parent, then we are done (because we respect the heap property).
- Otherwise, we need to “percolate up” the newly-inserted element recursively until we find a location that respects the heap property.
- The upward percolation is simple, we just recursively compare the new element with its parent. If the new element is smaller, then we swap it with its parent; otherwise we stop.

HEAP OPERATION: INSERT



HEAP OPERATION: INSERT

```
1  /**
2  * Insert item x, allowing duplicates.
3  */
4  void insert( const Comparable & x )
5  {
6      if( currentSize == array.size( ) - 1 )
7          array.resize( array.size( ) * 2 );
8
9      // Percolate up
10     int hole = ++currentSize;
11     Comparable copy = x;
12
13     array[ 0 ] = std::move( copy );
14     for( ; x < array[ hole / 2 ]; hole /= 2 )    // compare with its parent
15         array[ hole ] = std::move( array[ hole / 2 ] );
16     array[ hole ] = std::move( array[ 0 ] );
17 }
```

HEAP OPERATION: INSERT

- The upward percolation resembles climbing from the leaf to the root. Because the heap is full, the height of the tree is $O(\log(n))$. In this case, the time complexity for upward percolation is also $O(\log(n))$.

HEAP OPERATION: DELETION

- `delete()`: In the context of priority queue, we are only interested in deleting the element with the highest priority (not any one).
- We know that the element with the highest priority resides in the root. Our task is thus to remove the root.

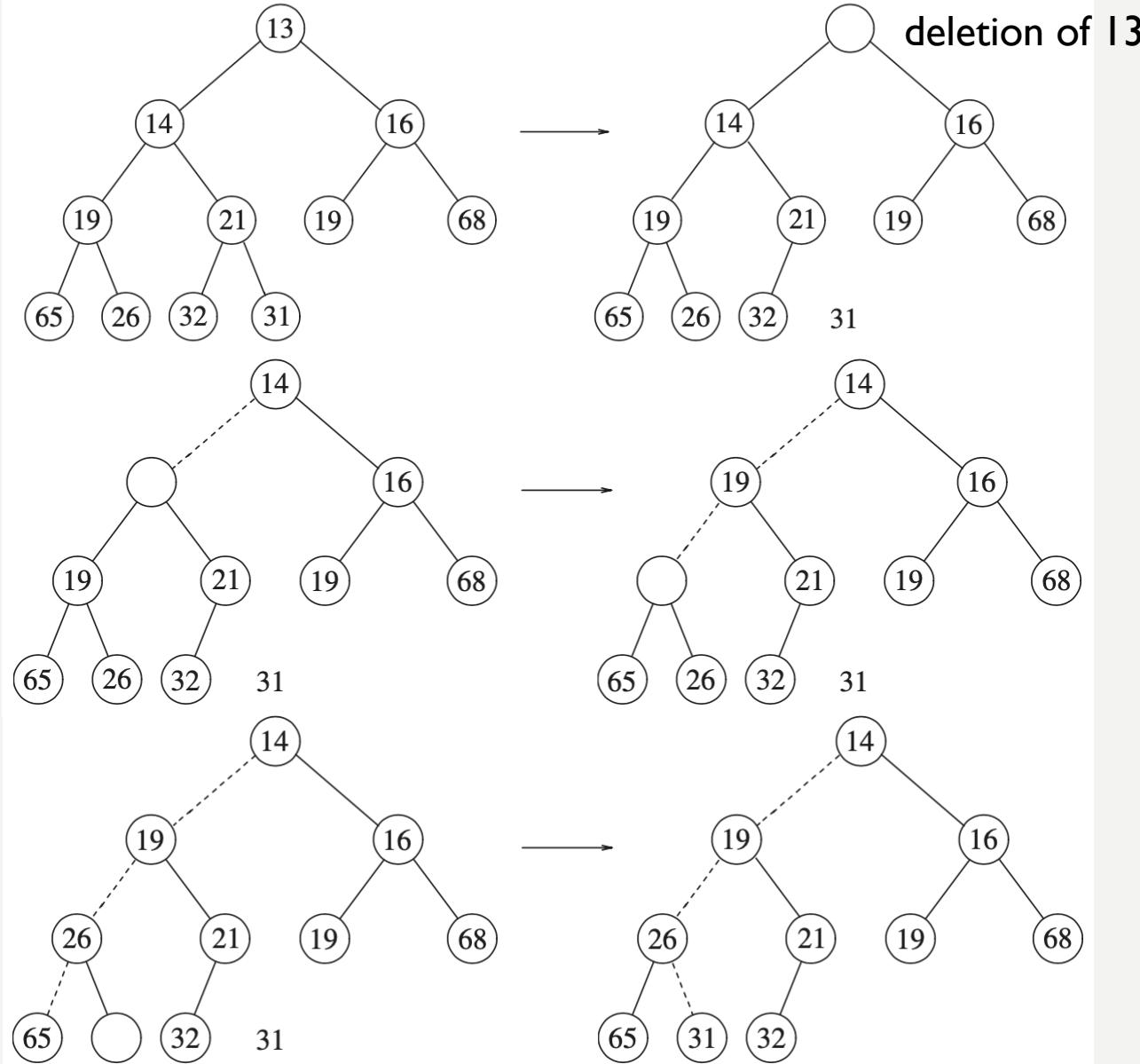
HEAP OPERATION: DELETION

- Once we remove the root, the heap is one element less and we should shrink the array size by 1.
- It expels the last element, and we put the last element in the root as its temporary home.
- However, doing so may violate the heap property (e.g., the last element could be larger than one of its children). In this case, we need to “percolate down” the last element to restore the heap property.

HEAP OPERATION: DELETION

- Among the three elements (the element we need to percolate down and its two children), the new parent should be the smallest element among the three.
- In this case, we compare the new element with its smaller child, if the smaller child is smaller than the new element, we swap them. Otherwise we stop.

HEAP OPERATION: DELETION



HEAP OPERATION: DELETION

```
28     /**
29      * Internal method to percolate down in the heap.
30      * hole is the index at which the percolate begins.
31      */
32     void percolateDown( int hole )
33     {
34         int child;
35         Comparable tmp = std::move( array[ hole ] );
36
37         for( ; hole * 2 <= currentSize; hole = child )
38         {
39             child = hole * 2;
40             if( child != currentSize && array[ child + 1 ] < array[ child ] ) // compares with the smaller child
41                 ++child;
42             if( array[ child ] < tmp )
43                 array[ hole ] = std::move( array[ child ] );
44             else
45                 break;
46         }
47         array[ hole ] = std::move( tmp );
48     }
```

HEAP OPERATION: OTHERS

- Other supported operations (with minor modification to the existing percolation algorithm) include:
 - **decreaseKey(p, d)**: decrease the element at position p (in the array) by an amount of d. This can be done by percolating the decreased element up.
 - **increaseKey(p, d)**: increase the element at position p by an amount of d. This can be done by percolating the increased element down.
 - **remove(p)**: remove the element at position p. This can be done by calling decreaseKey(p, -INF) and deleteMin() (which resembles dequeue()).

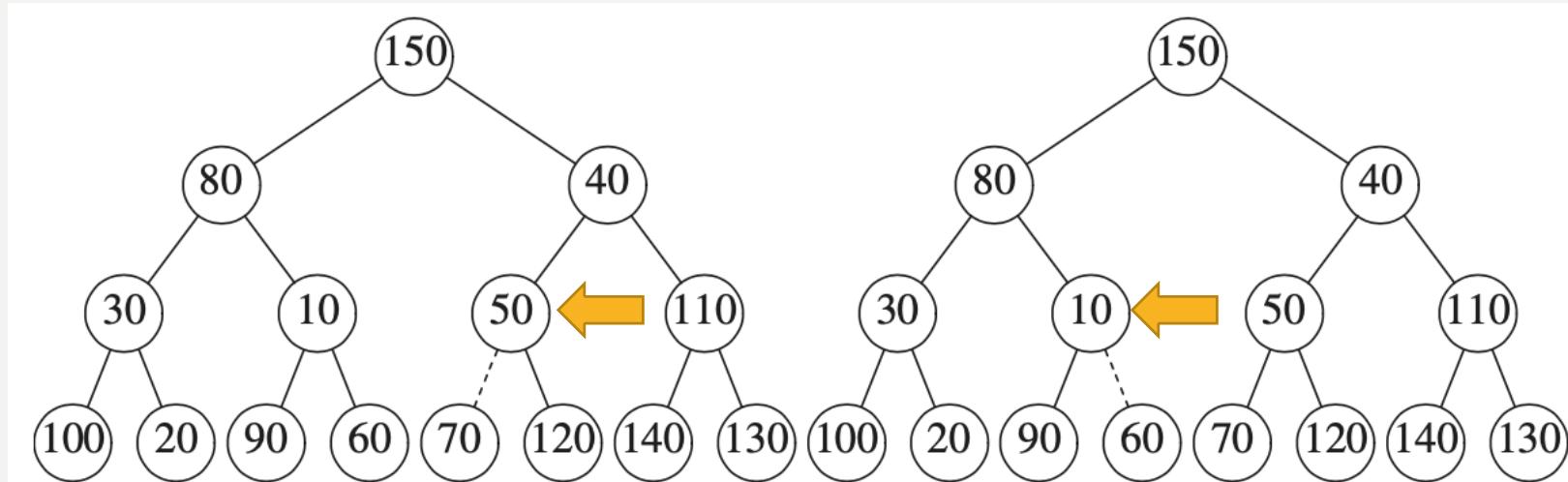
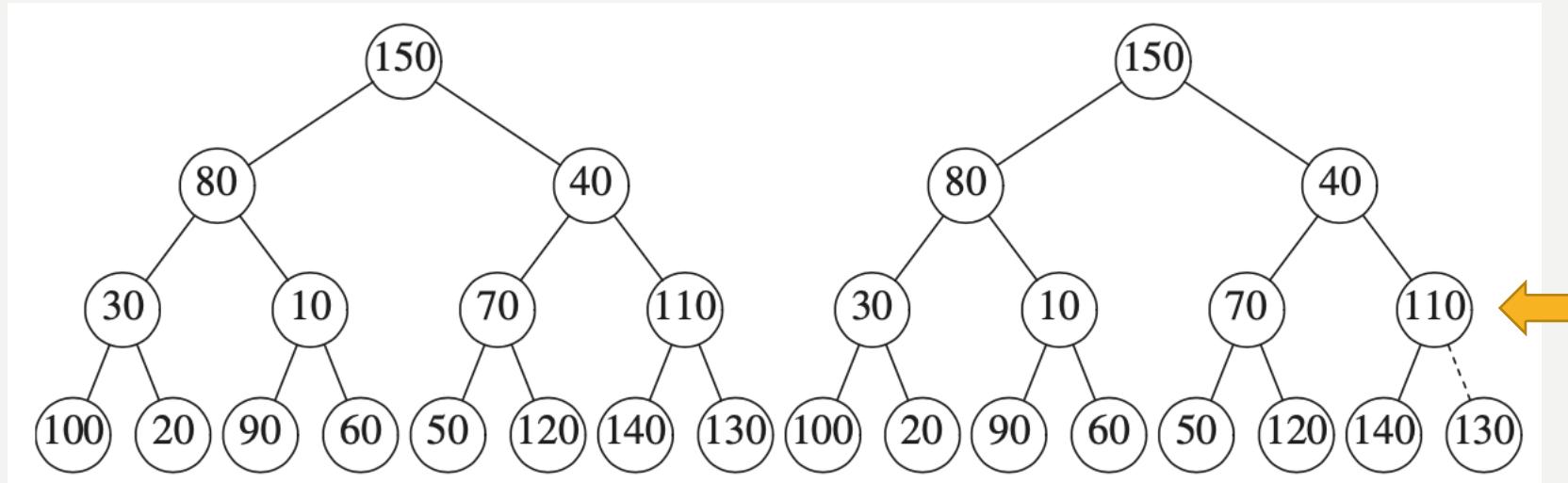
HEAP CONSTRUCTION

- Naïve way: we can insert the elements into the heap one-by-one. We know that each heap insertion can be done in $O(\log(n))$ time. In this case, inserting n elements would take $O(n\log(n))$ time.
- Can we do better?
 - Given the n elements (in arbitrary order), we wish to construct a heap for them in $O(n)$ time

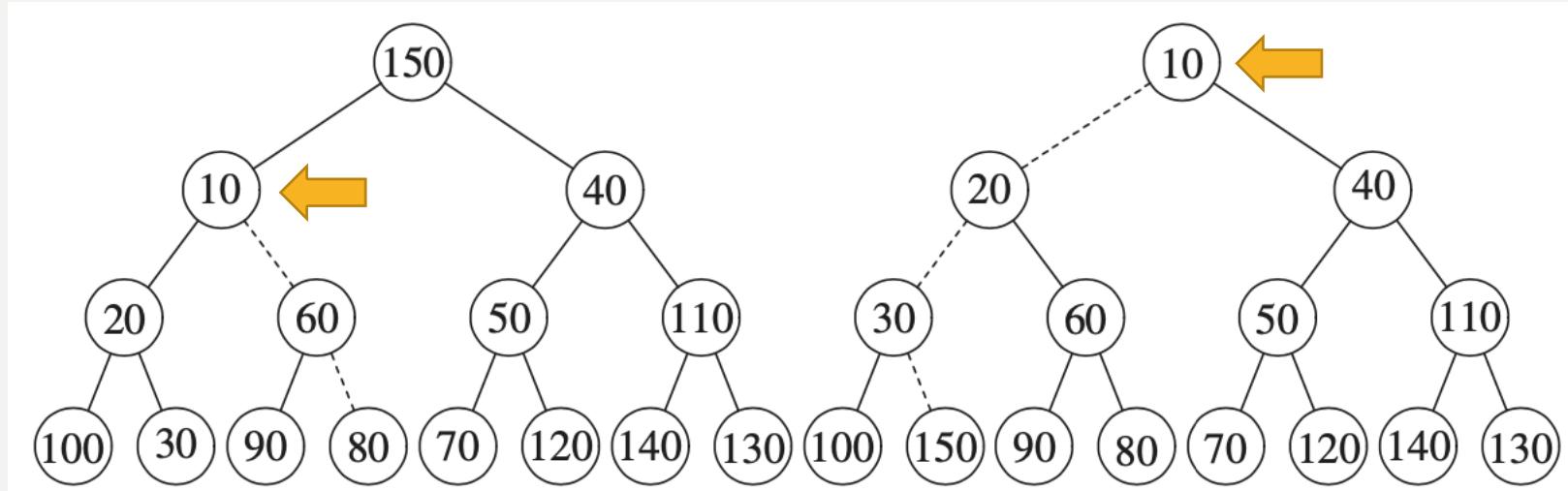
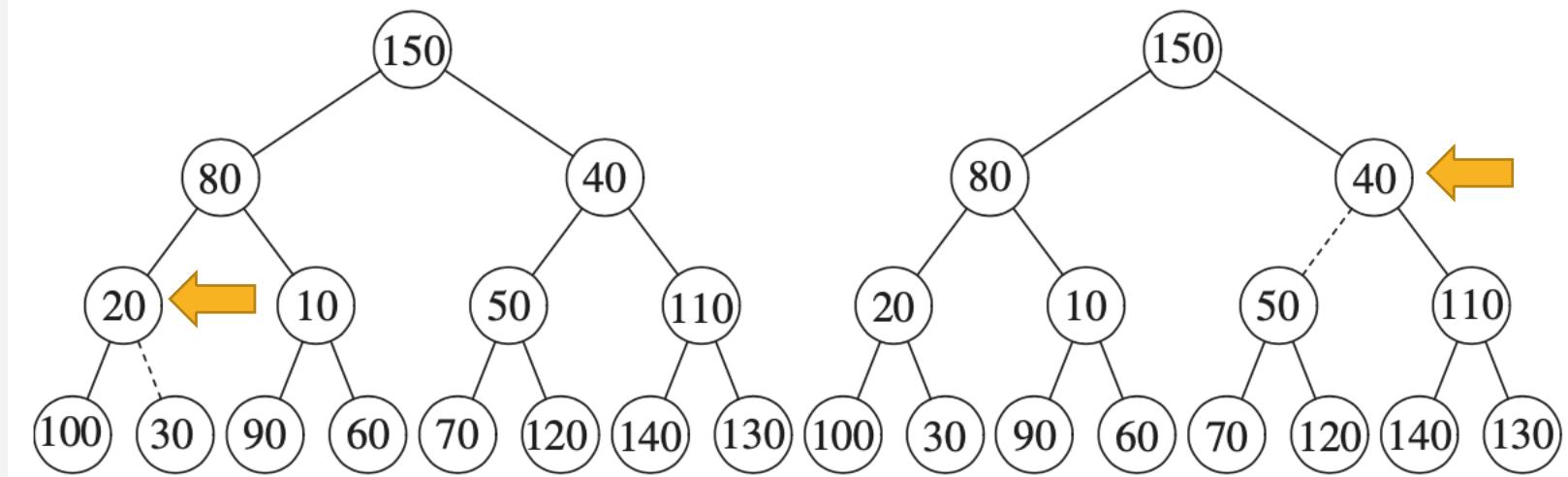
LINEAR TIME HEAP CONSTRUCTION

- The algorithm:
 - we copy all elements into the array (using $O(n)$ time); it indicates that we randomly place all elements into the binary tree
 - then, from the last element who is not a leaf to the root, we perform downward percolation (more detailed time analysis will follow)
 - done!
- This is correct because we have essentially percolate down all elements, and finally the resulted heap must have the heap property.

LINEAR TIME HEAP CONSTRUCTION



LINEAR TIME HEAP CONSTRUCTION



LINEAR TIME HEAP CONSTRUCTION

```
9      /**
10     * Establish heap order property from an arbitrary
11     * arrangement of items. Runs in linear time.
12     */
13    void buildHeap( )
14    {
15        for( int i = currentSize / 2; i > 0; --i )
16            percolateDown( i );
17    }
```

LINEAR TIME HEAP CONSTRUCTION

- Because heap is full, we expect that approximately half of the nodes are leaf; and we don't need to do anything to them.
- Recall that we perform downward percolation from the lower layers to the upper layers. While lower layers contain more nodes, they will also incur fewer steps for each downward percolation.
- Intuitively, we perform “quick” downward percolation for most of the nodes, and “slower” downward percolation for only few of the nodes. Overall our algorithm will be very efficient.
- Now we will go through the formal proof.

LINEAR TIME HEAP CONSTRUCTION (PROOF)

- Theorem: A completely full binary tree has $2^{h+1} - 1$ nodes, the sum of their heights is $2^{h+1} - 1 - (h + 1)$
 - Note that the total heights will serve as an upper bound for the downward percolation
 - That is, the total number of downward percolation will not exceed the total heights
 - If this is proved, then we can prove the $O(n)$ running time, because on average the number of percolation associated with each node is $O(1)$ (divide the total heights by the total number of nodes)

LINEAR TIME HEAP CONSTRUCTION (PROOF)

- Proof:
 - Note that we have 1 root, and its height is 2^h (the 0th layer).
 - In the next level (the 1st layer), we have 2 nodes, and their heights are $h - 1$.
 - In the next level (the 2nd layer), we have $2^2 = 4$ nodes, and their heights are $h - 2$
 -
 - In the i th layer, we have 2^i nodes, and the height of each node is $2^{h-1} - i$

LINEAR TIME HEAP CONSTRUCTION (HEAP)

- Proof cont.
 - the total height: $S = \sum_{i=0}^h 2^i(h - i) = h + 2(\textcolor{green}{h - 1}) + \textcolor{red}{4(h - 2)} + \textcolor{blue}{8(h - 3)} + \dots + 2^h(1)$
 - multiply it by 2: $2S = 2 \sum_{i=0}^h 2^i(h - i) = \textcolor{green}{2h} + \textcolor{red}{4(h - 1)} + \textcolor{blue}{8(h - 2)} + 16(h - 3) + \dots + 2 * 2^h(1)$
 - we notice that many of the terms will actually cancel out
 - if we subtract the first equation from the second, we will get: $S = 2 + 4 + 8 + \dots + 2^h - h = 1 + 2 + 4 + 8 + \dots + 2^h - (h + 1) = 2^{h+1} - 1 - (h + 1)$

MERGING HEAPS: MOTIVATION

- One limitation of the existing heap implementation is that it may take a lot of time to merge two heaps.
- Essentially we will construct a new heap using both of their elements, which requires a linear time as discussed above.
- Can we do better? Imagine that each heap respects the structural and heap properties, and we could develop a smarter way to merge them without disrupting individual heaps.
 - Several implementations that allow $O(\log(n))$ merge

MERGING HEAPS: MOTIVATION

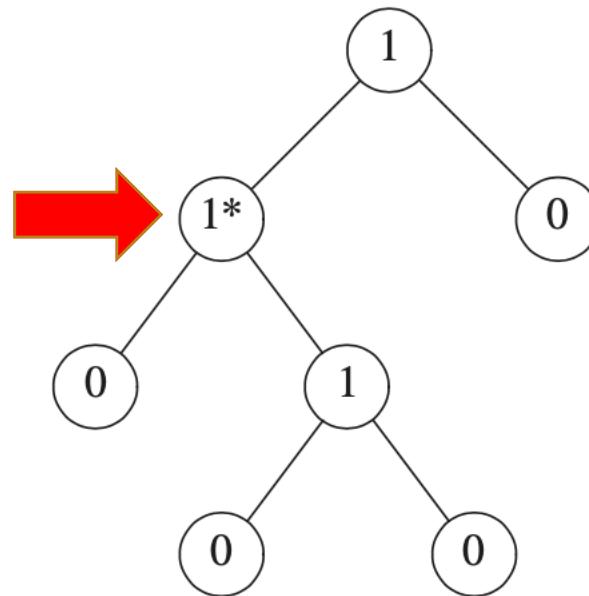
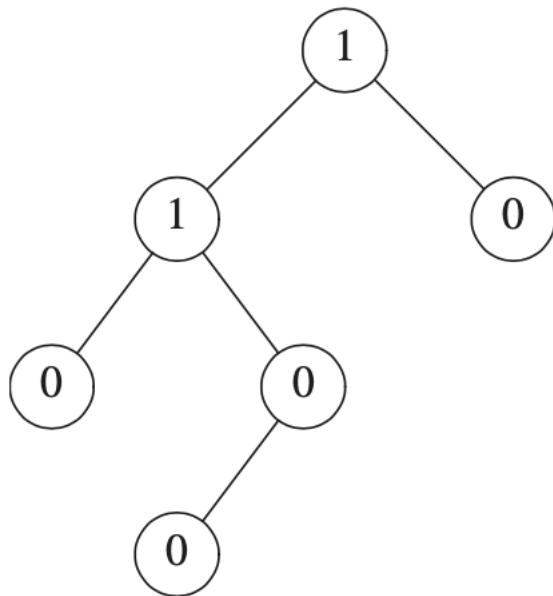
- Two enhanced heap data structures that support faster merge
 - leftist heap
 - binomial heap

LEFTIST HEAP

- The leftist heap has the same heap property, that is, the parent is smaller than its two children.
- However, the leftist heap has a different topology; it is not necessarily full.
- The leftist heap property:
 - define the null path length of a node as the shortest path length between itself and one of its descendent leaf
 - for each node in the leftist heap, the null path length of its left child must be at least as large as its right child

LEFTIST HEAP

- The left one is a leftist heap, while the right one is not.
- The null path length of the right child of the highlighted node is larger than the null path length of its left child.



LEFTIST HEAP: FUNDAMENTAL OPERATIONS

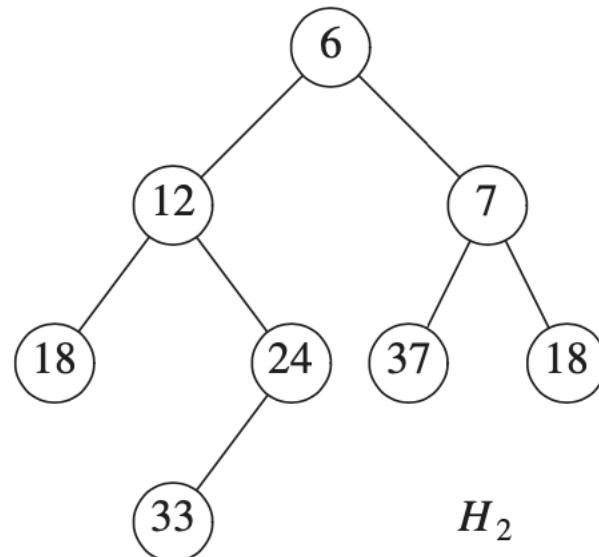
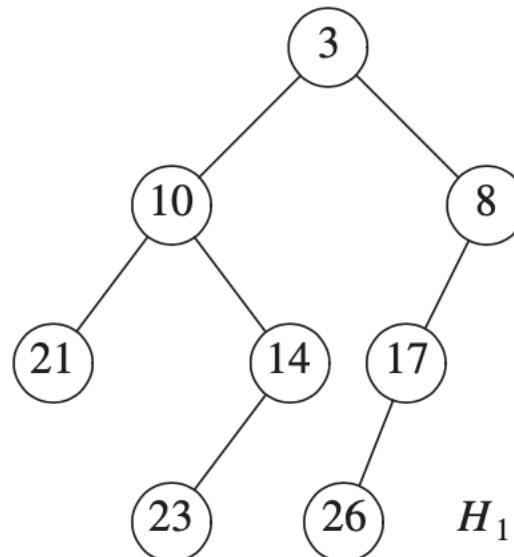
- Note that the leftist heap is still a heap. That is, it still needs to perform the fundamental top(), enqueue(), dequeue() operations efficiently (more specifically, top() in $O(1)$ time, enqueue() in $O(\log(n))$ time and dequeue() in $O(\log(n))$ time).
- The top() operation can be performed within $O(1)$ time through returning the root element.
- The enqueue() and dequeue() operations are performed through **merge()**:
 - enqueue(): **merge** the existing leftist heap with the leftist heap that contains the sole element to insert
 - dequeue(): remove the root, and **merge** the remaining two subtrees

LEFTIST HEAP: MERGE

- It is fair to say, that the fundamental operations of the leftist heap is **merge**.
- The merge will take two **recursive** steps:
 - merge the heap with a larger root with the right subtree of the heap with a smaller root
 - if the null path length of the right subtree is smaller than the null path length of the left subtree, swap the two subtrees

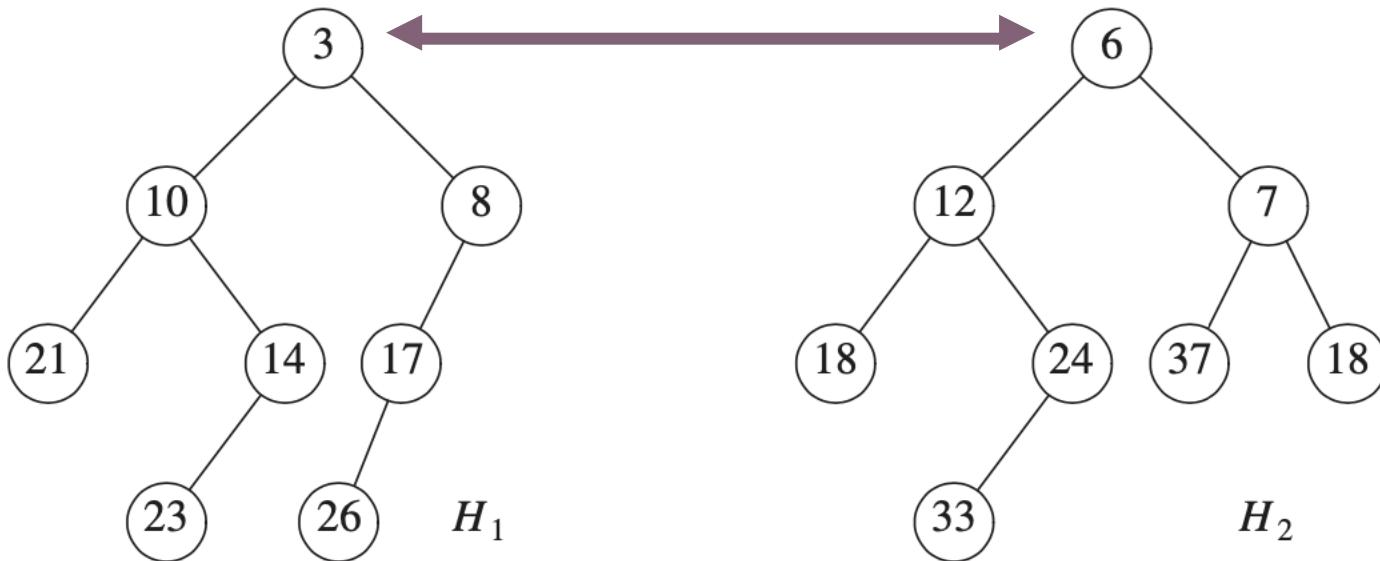
LEFTIST HEAP: MERGE

- imagine we are merging the following two subtrees



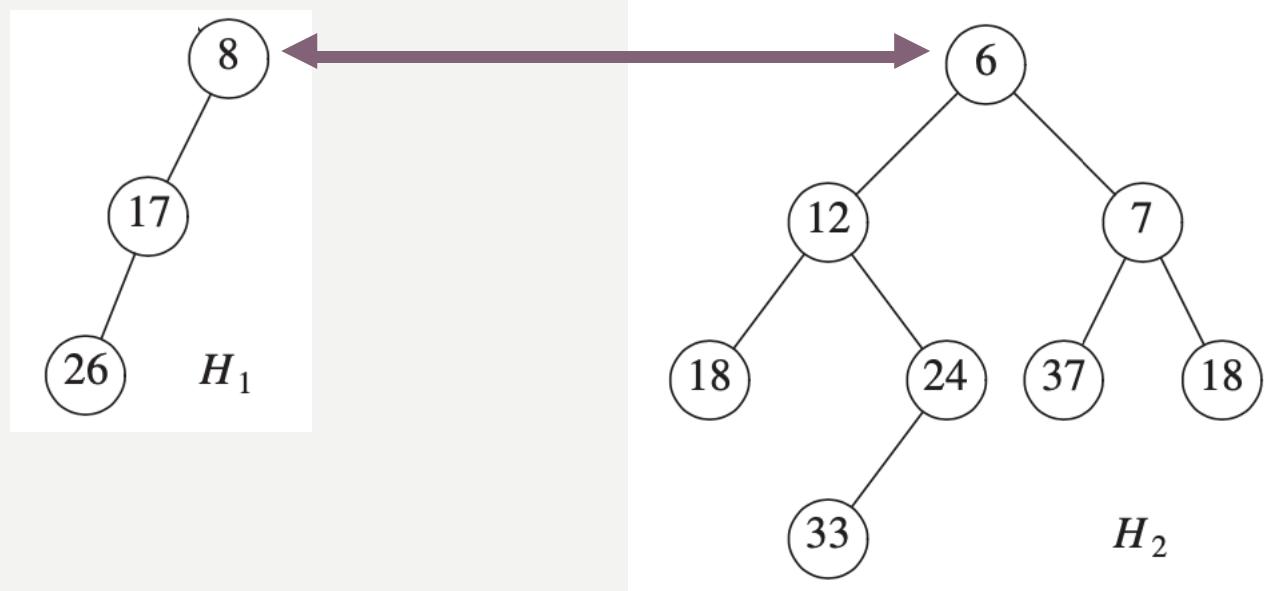
LEFTIST HEAP: MERGE

We compare the roots, the right heap has a larger root.
So, we merge it with the right subtree of the left heap.



LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.



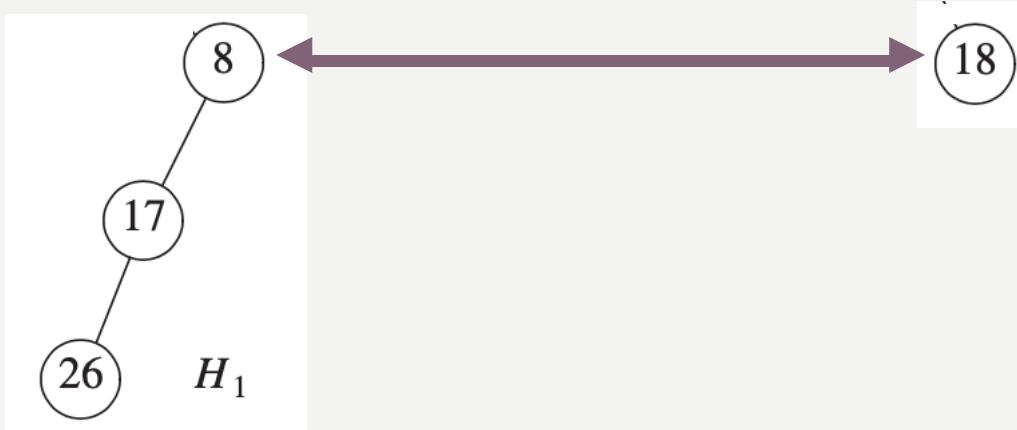
LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.



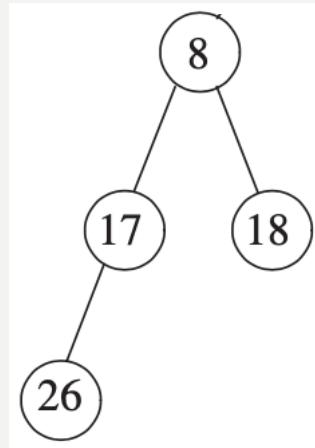
LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.



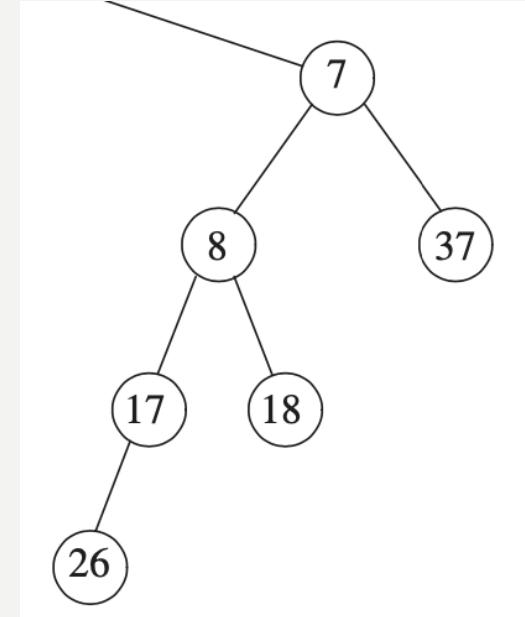
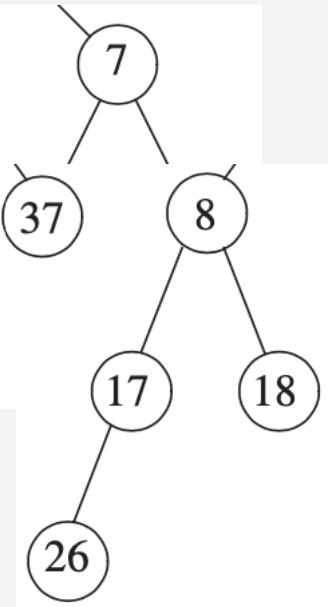
LEFTIST HEAP: MERGE

Now we can merge them!!!



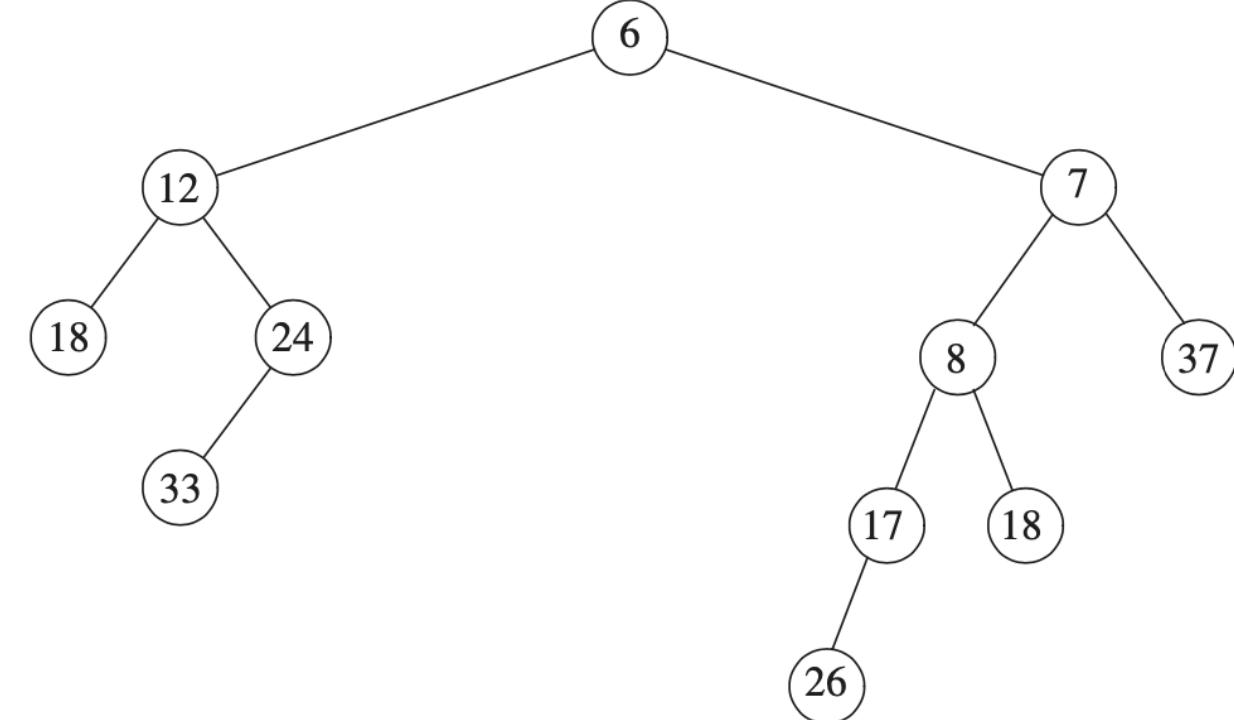
We have to check if the left subtree has a larger null path length.
Checked, no swapping is needed.

LEFTIST HEAP: MERGE



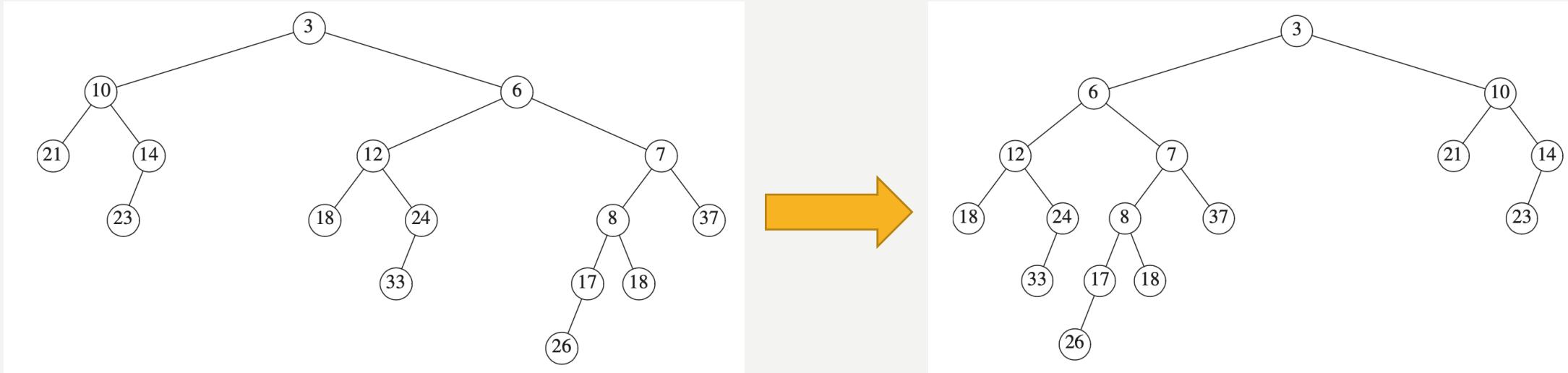
We have to check if the left subtree has a larger null path length.
No, so we swap them.

LEFTIST HEAP: MERGE



We have to check if the left subtree has a larger null path length.
Checked, no swapping is needed.

LEFTIST HEAP: MERGE



We have to check if the left subtree has a larger null path length.
No, swap them.
And we are done!

LEFTIST HEAP: MERGE

```
1  /**
2   * Internal method to merge two roots.
3   * Assumes trees are not empty, and h1's root contains smallest item.
4   */
5 LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
6 {
7     if( h1->left == nullptr )    // Single node
8         h1->left = h2;          // Other fields in h1 already accurate
9     else
10    {
11        h1->right = merge( h1->right, h2 );
12        if( h1->left->npl < h1->right->npl )
13            swapChildren( h1 );
14        h1->npl = h1->right->npl + 1;
15    }
16    return h1;
17 }
```

// key assumption

// boundary case: we attach h2 as
the left child to ensure the leftist
property

// recursive merge in deeper levels

// swap if necessary
// update null path length

LEFTIST HEAP: MERGE

- Time complexity? We expect $O(\log(n))$.
- If we can prove the time complexity, we can indirectly show that enqueue() and dequeue() will both take $O(\log(n))$ time. This is because each of them will only incur a single merge operation.
- Recall the merge process, all operations (comparison and swapping) are performed on the right subtree.
- Therefore, we would like to show the time complexity through bounding the right subtree height.

LEFTIST HEAP: MERGE

- Theorem: A leftist tree with r nodes on the right path must contain at least $2^r - 1$ nodes.
- Proof (informal): Intuitively, according to the leftist tree property (the null path length of the left child is larger than the null path length of the right child), the tree should be skew to the right (i.e., higher left subtrees and shorter right subtrees). In this case, if the right path has r nodes, then all other paths must have at least r nodes. The tree is then with a height of r ; and such a tree clearly has at least $2^r - 1$ nodes.

LEFTIST HEAP: MERGE

- Proof (formal): The proof is by induction. If $r = 1$, there must be at least one tree node. Otherwise, suppose that the theorem is true for $1, 2, \dots, r$. Consider a leftist tree with $r + 1$ nodes on the right path. Then the root has a right subtree with r nodes on the right path, and a left subtree with at least r nodes on the right path (otherwise it would not be leftist). Applying the inductive hypothesis to these subtrees yields a minimum of $2^r - 1$ nodes in each subtree. This plus the root gives at least $2^{r+1} - 1$ nodes in the tree, proving the theorem.

LEFTIST HEAP

- In summary, the merge of the leftist heap will take $O(\log(n))$ time.
- The element with the highest priority will reside on the root, and retrieving that element will only take $O(1)$ time.
- `enqueue()` can be done through merging the existing leftist heap with a leftist heap that contains only the element to be inserted. `dequeue()` can be done through deleting the root and subsequently merging the remaining left and right subtrees. Both operations will take $O(\log(n))$ time.
- Unfortunately, unlike the regular heap, leftist heap cannot be implemented using array (because the leftist heap is not full). We will need to use pointers for the implementation, which is less computational and space efficient. **So, only use leftist heap if merge is a frequent operation.**

BINOMIAL QUEUE

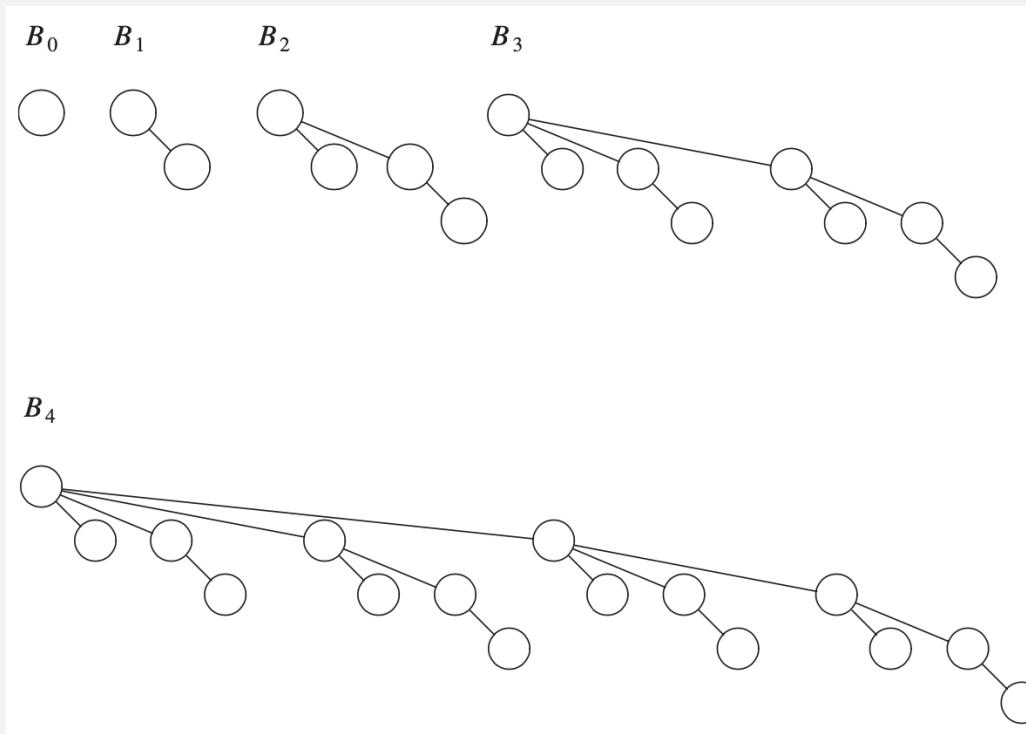
- Binomial queue is another data structure that supports fast merge operation.
- It still requires $O(1)$ time for retrieving the element with the highest priority, and $O(\log(n))$ for merge, enqueue(), and dequeue() in the worst-case scenario.
 - but it only requires, on average, constant time for enqueue() (**this is better than the leftist heap**)

BINOMIAL QUEUE

- A binomial queue is a set of trees (also called forest).
- The trees contain different number of nodes. The different numbers of nodes contained in different trees correspond to the different exponents of 2. For example, there could be a tree that contains 1 node (2^0), another tree that contains 16 nodes (2^4), another tree that contains 64 nodes (2^6)... However, we will not have any tree that contains, e.g., 6 nodes, which is not an exponent of 2. Hence, each tree is called a binomial tree. (And it is also where the name binomial queue comes from.)
- The sets of trees to use depends on the number of elements in the queue, which is broken down into the sum of exponents of 2 that will be used to determine the sets of tree in use. For example, if we have 381 nodes, it can be broken down into $1+4+8+16+32+64+256$. So, we will use a tree with 1 node, a tree with 4 nodes, a tree with 8 nodes, ..., and a tree with 256 nodes.

BINOMIAL QUEUE

- The trees are defined recursively. Let B_i be the tree that contains 2^i nodes. B_0 is a one-node tree. B_k is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree B_{k-1} .
- Each binomial tree has the heap property: the parent is smaller (with a higher priority) than its children.

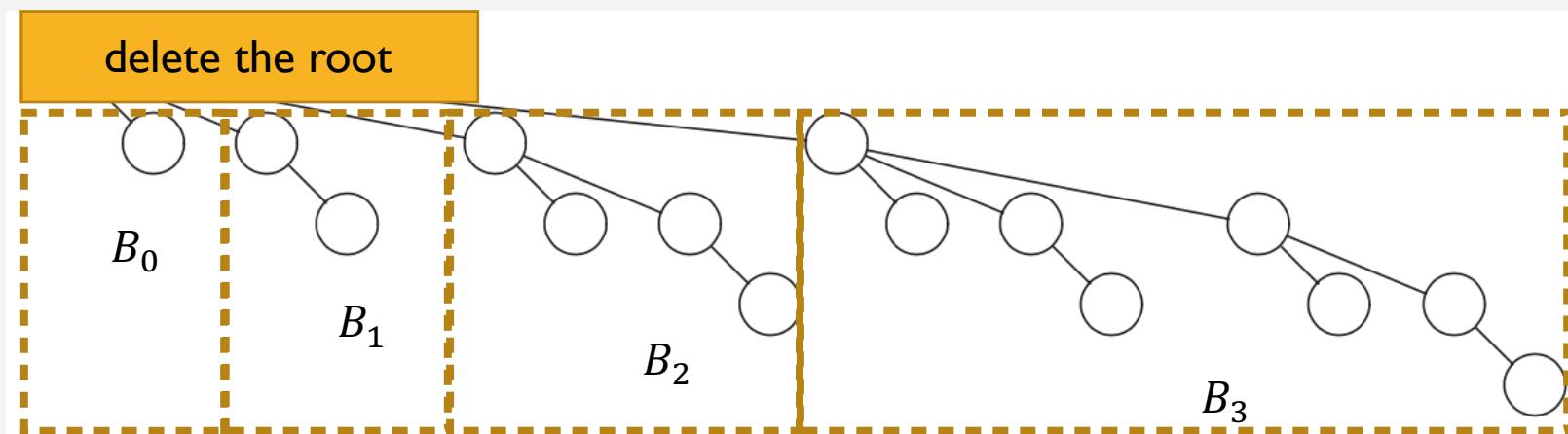
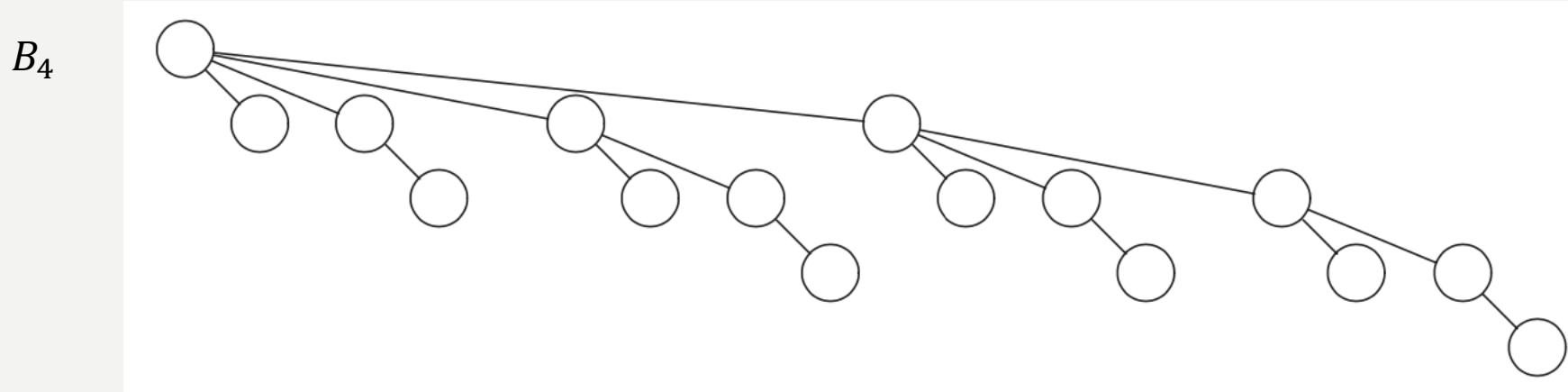


BINOMIAL QUEUE: OPERATIONS

- We claim that we can access the element with the highest priority in $O(1)$ time. To achieve this, we can simply set up a pointer that points to the smallest root among all binomial trees. (Keep in mind that we will need to update this pointer when insertion/deletion happens.)
- Similar to leftist heap, enqueue() and dequeue() of the binomial queue can also be performed through using merge():
 - enqueue(): merge the existing binomial queue with a binomial queue that contains a single node
 - dequeue(): deleting the root of a binomial tree will form a binomial queue, we can then merge the existing binomial queue (without the binomial tree under operation) and the newly formed (by deleting the node) binomial queue

BINOMIAL QUEUE: OPERATIONS

- Deletion: example



BINOMIAL QUEUE: OPERATIONS

- Similar to leftist heap, binomial heap also takes merge as its essential operation.
- The merge process of binomial queue resembles simple arithmetic addition of two binary numbers.

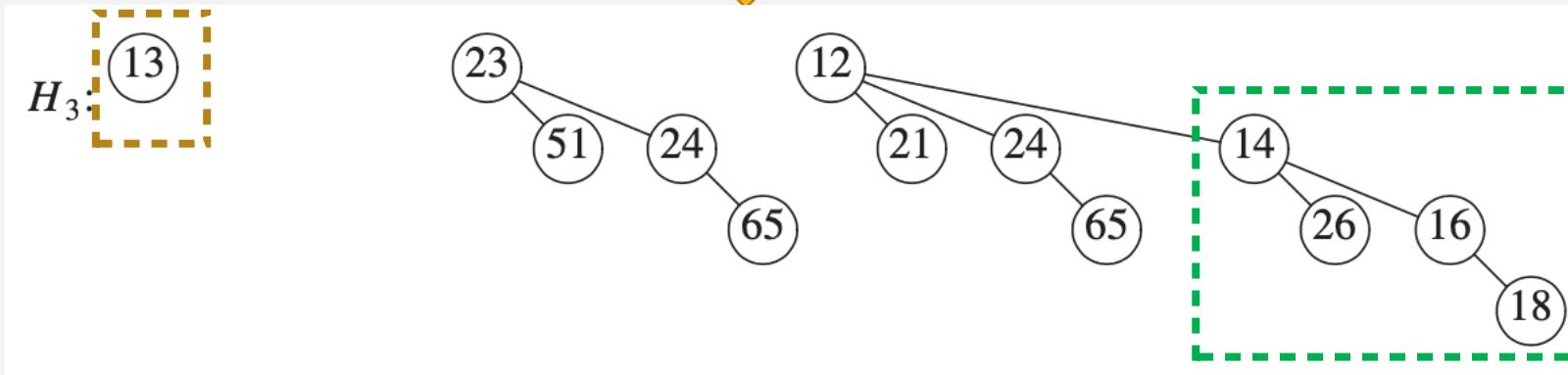
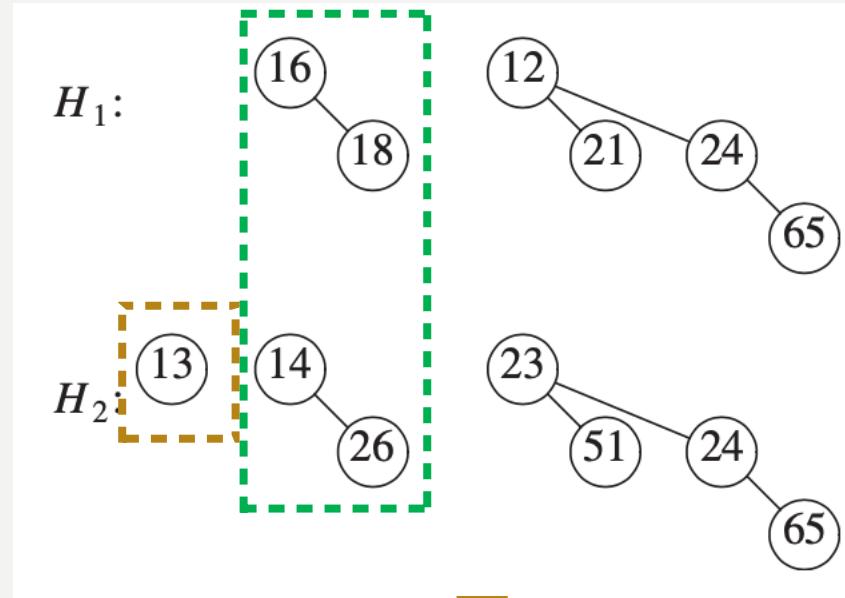
BINOMIAL QUEUE: OPERATIONS

- For example, let's say we have two binomial queues, one with 15 nodes and another with 12 nodes. We can decompose the first queue (say A) as $1+2+4+8$ (or $B_0^A + B_1^A + B_2^A + B_3^A$), and the second queue (say B) as $B_2^B + B_3^B$.
- The merge is essentially looking to construct a new queue $C = A + B = B_0^A + B_1^A + B_2^A + B_3^A + B_2^B + B_3^B$.

BINOMIAL QUEUE: OPERATIONS

- Recall the recursive definition of binomial tree: it says that we can merge two B_{k-1} s to get a new B_k . **To retain the heap property, we attach the binomial tree with a larger root to the one with a smaller root.**
- Let's go back to the original problem of computing $C = B_0^A + B_1^A + B_2^A + B_3^A + B_2^B + B_3^B$.
- Starting from the smallest trees, we notice that we only have one B_0 , and in this case we will directly take this tree, B_0^A , into our new queue ($B_0^A = B_0^C$). Similarly, we also take B_1^A directly ($B_1^A = B_1^C$). Next, we notice that we have two B_2 s, and we will merge them into one B_3 , say B_3^C . Finally, we also have two B_3 s and we will merge them into one B_4 , say B_4^C .
- Finally, we will have $C = B_0^C + B_1^C + B_3^C + B_4^C$. We can double check the total number of elements in the new queue, we have $1+2+8+16=27$ elements, which equals to the total number of elements in the original queues ($27=15+12$).

BINOMIAL QUEUE: OPERATIONS



BINOMIAL QUEUE: OPERATIONS

- Merge time complexity: $O(\log(n))$
 - we know that the binomial queue that contains n elements has $\log(n)$ binomial trees, because each tree is at least twice larger than the previous tree
 - merging each pair of binomial trees only takes $O(1)$ time (specifically, one root comparison, and one attachment)
- Because `enqueue()` and `dequeue()` can be done through a single call of the merge function, both of them will also require $O(\log(n))$ time.

BINOMIAL QUEUE: LINEAR-TIME INSERTION

- Remember at the beginning, we claim that `enqueue()` can be done in $O(1)$ time on average. However, our previous analysis shows that it needs $O(\log(n))$ time in the worst case scenario.
- To see the $O(1)$ time average complexity, we need to model the merge process as a Bernoulli process. Just like flipping a coin, we consider that the chance the existing binomial queue contains a binomial tree with a specific size as a random variable with 50-50 chance. That is, if we do not know the the size of the binomial queue, we will assume that it has 50% chance to contain B_0 , 50% chance to contain B_1, \dots etc.

BINOMIAL QUEUE: LINEAR-TIME INSERTION

- Consider the enqueue() process, where we recursively try to merge the single-node binomial tree with the element to insert with the existing binomial trees. The merge process will stop if the existing binomial queue does not contain a specific binomial tree.
- For example, consider a binomial queue $B_0B_1B_3B_4$, the insertion will terminate after two steps because the existing tree does not contain B_2 .
- The number of steps we need for the insertion thus equals to the number of continuously-double-sized binomial trees that the queue has.

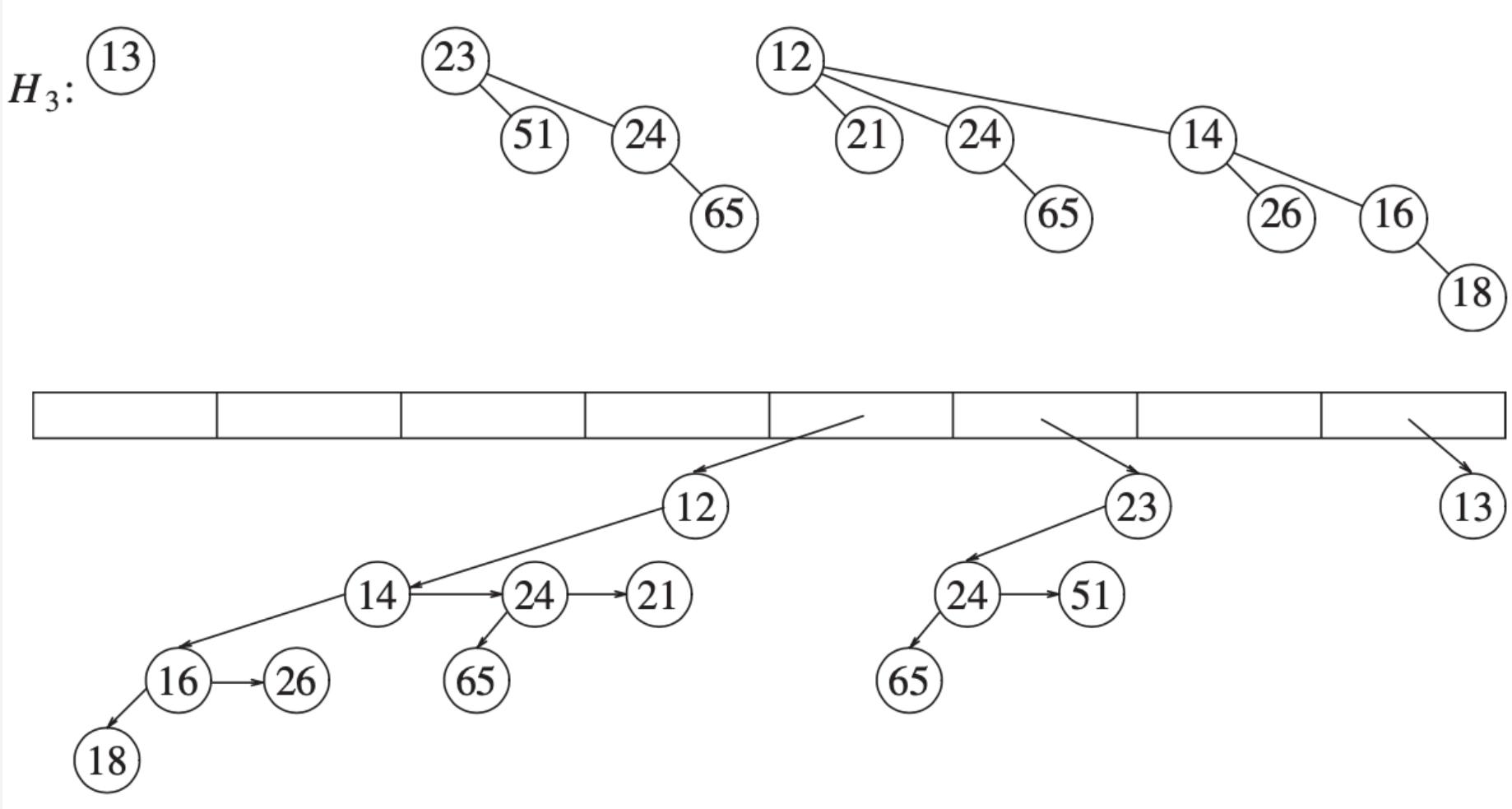
BINOMIAL QUEUE: LINEAR-TIME INSERTION

- If we consider that the existing binomial queue contains a specific binomial tree as getting a head in a coin-flip (both has 50% chance), then the number of merge steps we need will correspond to the number of heads we can get before getting a tail.
- The distribution of this random variable is modeled by the **negative binomial distribution**.
- The negative binomial distribution has an expected value of $p/(1 - p)$, where p is the chance of getting a head in the experiment. In our case, $p = 0.5$ and the expected value is 1. It says we expect to see one head before seeing one tail (which makes a general sense for a fair coin toss).
- Therefore, we can expect to only trigger 1 merge operation for each insertion.

BINOMIAL QUEUE: SUMMARY

- Binomial queue contains a set of binomial trees, each with different sizes that are exponents of 2.
- A binomial queue with n elements contains $O(\log(n))$ binomial trees. This is the fundamental observation to establish its related time complexity.
- Binomial queue supports:
 - `top()`: $O(1)$
 - `enqueue`: $O(\log(n))$ in the worst-case scenario, $O(1)$ on average (note that when we merge, we also need to update the MIN pointer if necessary; the MIN pointer helps us to locate the element with the highest priority in a constant time)
 - `dequeue()`: $O(\log(n))$
 - `merge()`: $O(\log(n))$

BINOMIAL QUEUE: IMPLEMENTATION



BINOMIAL QUEUE: IMPLEMENTATION

```
26     private:
27         struct BinomialNode
28     {
29             Comparable     element;
30             BinomialNode *leftChild;
31             BinomialNode *nextSibling;
32
33             BinomialNode( const Comparable & e, BinomialNode *lt, BinomialNode *rt )
34                 : element{ e }, leftChild{ lt }, nextSibling{ rt } { }
35
36             BinomialNode( Comparable && e, BinomialNode *lt, BinomialNode *rt )
37                 : element{ std::move( e ) }, leftChild{ lt }, nextSibling{ rt } { }
38         };
39
40         const static int DEFAULT_TREES = 1;
41
42         vector<BinomialNode *> theTrees; // An array of tree roots // the array containing pointers to the subtrees
43         int currentSize; // Number of items in the priority queue
```

BINOMIAL QUEUE: IMPLEMENTATION

```
1      /**
2       * Return the result of merging equal-sized t1 and t2.
3       */
4      BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
5      {
6          if( t2->element < t1->element )           // compare the root; attach the subtree with a
7              return combineTrees( t2, t1 );          larger root to the one with a smaller root to
8          t2->nextSibling = t1->leftChild;         maintain the heap property
9          t1->leftChild = t2;
10         return t1;
11     }
```

BINOMIAL QUEUE: IMPLEMENTATION

```
22     BinomialNode *carry = nullptr;
23     for( int i = 0, j = 1; j <= currentSize; ++i, j *= 2 )
24     {
25         BinomialNode *t1 = theTrees[ i ];
26         BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ]
27                                     : nullptr;
28         int whichCase = t1 == nullptr ? 0 : 1; // whether the first queue contains the subtree
29         whichCase += t2 == nullptr ? 0 : 2;    // whether the second queue contains the subtree
30         whichCase += carry == nullptr ? 0 : 4; // whether exists a carrying-over subtree
```

BINOMIAL QUEUE: IMPLEMENTATION

```
32         switch( whichCase )          45             case 3: /* this and rhs */  
33         {                           46                 carry = combineTrees( t1, t2 );  
34             case 0: /* No trees */      47                 theTrees[ i ] = rhs.theTrees[ i ] = nullptr;  
35             case 1: /* Only this */    48                 break;  
36                 break;  
37             case 2: /* Only rhs */     49  
38                 theTrees[ i ] = t2;    50  
39                 rhs.theTrees[ i ] = nullptr; 51  
40                 break;  
41             case 4: /* Only carry */   52  
42                 theTrees[ i ] = carry; 53  
43                 carry = nullptr;    54  
44                 break;           55  
45         }                           56  
46         case 5: /* this and carry */ 57             case 6: /* rhs and carry */ 58                 carry = combineTrees( t2, carry );  
47             carry = combineTrees( t1, carry ); 59                 rhs.theTrees[ i ] = nullptr;  
48             theTrees[ i ] = nullptr;        60                 break;  
49             break;                   61             case 7: /* All three */     62                 theTrees[ i ] = carry;  
50         }                           63                 carry = combineTrees( t1, t2 );  
51         case 7: /* All three */     64                 rhs.theTrees[ i ] = nullptr;  
52             theTrees[ i ] = carry;    65                 break;  
53             break;               66         }  
54     }  
55 }
```

C++ STL PRIORITY QUEUE

- std::priority_queue
 - void push(const Object & x);
 - const Object & top() const;
 - void pop();
 - bool empty();
 - void clear();

SUMMARY

- Priority queue: an extension of the queue ADT with user-defined feature to determine priority
- Implementations
 - heap: array implementation, fast and simple, but does not support merge well (needs $O(n)$) time for merge
 - leftist heap: supports $O(\log(n))$ merge, but requires pointers that makes it slower and more memory-intensive
 - binomial queue: also supports $O(\log(n))$ merge, further improves leftist heap with an expected $O(1)$ enqueue() time.

Time complexity analysis

Ben Liu

ben_0522@ku.edu

Definition

- Time complexity analysis is also called, running time (complexity) analysis. It refers to the analysis of the running time of an **algorithm** with respect to different **size** of input.
- Algorithm is a sequence of pre-defined statements for some purpose, the running time of the algorithm depends on:
 1. The environment (language, hardware, ...)
 2. The implementation (bubble sort vs merge sort)
 3. The input size (sort 10 items vs 1000 items)

RAM Model

- RAM is an ideal model used to analyze the running time of an algorithm without taking into consideration the power of environment:
 1. Each basic operation takes constant time;
 2. Exactly one statement can be executed at one time;
 3. Any datum can be stored in one memory block;
 4. Any stored datum can be accessed at constant time;

Type of analysis

- Exact analysis
- Approximation analysis
- Asymptotic analysis

Example 1 – exact analysis

```
4 int a = 5;
5 int d = 2;
6 int sum = 0;
7 int k = 100;
8 for(int i = 0; i <= k; ++i)
{
    int a_prime = a + i*d;
    sum = sum + a_prime;
}
```

- Operations:
 1. Assign, (line 4, line 5, ...) takes t_1 ;
 2. Compare, (line 8), takes t_2 ;
 3. Addition, (line 8, line 10) takes t_3 ;
 4. Multiplication, (line 10) takes t_4 ;
- Counting operations:
 1. Line 4 to line 7 were executed once;
 2. for-loop was executed 101 times;
 3. Note only initialize once.

Example 1 – exact analysis

```
4 int a = 5;
5 int d = 2;
6 int sum = 0;
7 int k = 100;
8 for(int i = 0; i <= k; ++i)
{
    int a_prime = a + i*d;
    sum = sum + a_prime;
}
```

- So, final counts of operations
 1. 308 assign;
 2. 102 compare (think of why?);
 3. 303 addition;
 4. 101 multiplication;
- All together:
$$T = 308t_1 + 102t_2 + 303t_3 + 101t_4$$

Example 1 – approximation analysis

```
4 int a = 5;
5 int d = 2;
6 int sum = 0;
7 int k = 100;
8 for(int i = 0; i <= k; ++i)
9 {
10     int a_prime = a + i*d;
11     sum = sum + a_prime;
12 }
```

- Operation blocks:
 1. Initialization , line 4, to line 7;
 2. for-loop, line 8 to line 12;
- Note that the running time of for-loop dominates Initialization. So it's reasonable for us to use the running time of for-loop to approximate the running time of the algorithm.

Example 1 – approximation analysis

```
4 int a = 5;
5 int d = 2;
6 int sum = 0;
7 int k = 100;
8 for(int i = 0; i <= k; ++i)
9 {
10     int a_prime = a + i*d;
11     sum = sum + a_prime;
12 }
```

- Now:

$$T = 304t_1 + 102t_2 + 303t_3 + 101t_4$$

- Before:

$$T = 308t_1 + 102t_2 + 303t_3 + 101t_4$$

- Not much difference, but it will be much easier for us, especially when the algorithm has tons of operation blocks.

Asymptotic analysis

- Asymptotic analysis is more useful when the input size changes, but here in example 1, the algorithm in example 1 doesn't accept input, or we can say the input size is constant.
- Let's see the definition of asymptotic analysis.

Asymptotic analysis

- In mathematics, asymptotic analysis studies the behavior of function $f(n)$ when n is very large.
- There are different notations of asymptotic, the following 3 are most used:
 1. Big-O, we focus on Big-O through this tutorial;
 2. Little-o;
 3. Big- Ω ;
 4. Little- ω ;
 5. Big- Θ ;

Asymptotic analysis

- Definition of Big-O:
for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in O(g(n))$ if and only if $f(n) \leq k \cdot g(n)$ when $n \geq n_0$. (Upper bound)
- Note that upper bound doesn't have to be tight, for example:
given $f(n) = n, g(n) = n^2, k = 1, n_0 = 1,$
apparently for $n \geq n_0, f(n) \leq k \cdot g(n)$, or for $n \geq 1, n \leq 1 \cdot n^2$, so $n \in O(n^2)$.
- But,
if $g(n) = n^3$, we still have for $n \geq 1, n \leq 1 \cdot n^3$, so $n \in O(n^3)$ as well.

Asymptotic analysis

- Definition of Little-o:

for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in o(g(n))$ if and only if $f(n) < k \cdot g(n)$ when $n \geq n_0$. (Tight upper bound)

- Definition of Big- Ω :

for any given function $f(n)$, if there exists two constants k, n_0 and another function $g(n)$, we say that $f(n) \in \Omega(g(n))$ if and only if $f(n) \geq k \cdot g(n)$ when $n \geq n_0$. (Lower bound)

Asymptotic analysis

- Definition of Little- ω :

for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in \omega(g(n))$ if and only if $f(n) > k \cdot g(n)$ when $n \geq n_0$. (Tight lower bound)

- Definition of Big- Θ :

for any given function $f(n)$, if there exists three constants k_1, k_2, n_0 and another function $g(n)$, we say that $f(n) \in \Theta(g(n))$ if and only if $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ when $n \geq n_0$. (Tight bound)

- Let's move on another example, sequential search.

Example 2

```
8 // A is an array of N integer items,
9 // k is the value we are looking for.
10 bool sequentialSearch(const std::vector<int> & A, int k)
11 {
12     if(A.empty())
13         return false;
14     for(int i = 0; i < A.size(); ++i)
15     {
16         if(A[i] == k) return true;
17     }
18     return false;
19 }
```

- Operations:
 1. empty check, t_1 ;
 2. return, t_2 ;
 3. assign, t_3 ;
 4. compare $<$, t_4 ;
 5. get size, t_5 ;
 6. addition, t_6 ;
 7. compare $==$, t_7 ;
- Dominant statement, for-loop.

Example 2

```
8 // A is an array of N integer items,
9 // k is the value we are looking for.
10 bool sequentialSearch(const std::vector<int> & A, int k)
11 {
12     if(A.empty())
13         return false;
14     for(int i = 0; i < A.size(); ++i)
15     {
16         if(A[i] == k) return true;
17     }
18     return false;
19 }
```

- Trivial case, if A is empty, the for-loop will be executed 0 time.
- Best case, if the first item of A is the item we are looking for, the for-loop will be executed 1 time.
- Worst case, if the last item of A is the item we are looking for, the for-loop will be executed N times.

Example 2

```
8 // A is an array of N integer items,
9 // k is the value we are looking for.
10 bool sequentialSearch(const std::vector<int> &A, int k)
11 {
12     if(A.empty())
13         return false;
14     for(int i = 0; i < A.size(); ++i)
15     {
16         if(A[i] == k) return true;
17     }
18     return false;
19 }
```

- Average case: assuming each item has equal probability p to be the item we are looking for, the for-loop is expected to be executed Np times.
- Which case should we use?

Example 2 - approximation analysis

```
8 // A is an array of N integer items,
9 // k is the value we are looking for.
10 bool sequentialSearch(const std::vector<int> & A, int k)
11 {
12     if(A.empty())
13     return false;
14     for(int i = 0; i < A.size(); ++i)
15     {
16         if(A[i] == k) return true;
17     }
18     return false;
19 }
```

- In the worst case:

$$T = (t_3 + t_4 + t_5 + t_6) \times N + t_3 + t_7$$

Example 2 - approximation analysis

$$T = (t_3 + t_4 + t_5 + t_6) \times N + t_3 + t_7$$

Let's assume:

$$a = (t_3 + t_4 + t_5 + t_6)$$

$$b = t_3 + t_7$$

Then,

$$T = aN + b$$

This is a function of N , so,

$$T(N) = aN + b$$

Example 2 - asymptotic analysis

- Given $T(N) = aN + b$, let's prove $T(N) \in O(N)$.

Let $g(N) = N$, $k = a + b$, $n_0 = 1$,

Obviously, for $N \geq 1$, which is $N \geq n_0$,

$$b \leq bN$$

$$aN + b \leq aN + bN$$

$$T(N) \leq aN + bN$$

$$T(N) \leq (a + b)N$$

$$T(N) \leq (a + b)g(N)$$

$$T(N) \leq kg(N)$$

So, we proved that for $N \geq 1$, $T(N) \in O(N)$.

Example 3

```
2 // A is an array of N integer items, sorted in non-descending order.
3 // k is the value we are looking for.
4 bool binarySearch(const std::vector<int> &A, int k)
5 {
6     if(A.empty())
7         return false;
8     int l = 0, h = A.size() - 1;
9     int m = 0;
10    while(l < h)
11    {
12        m = floor( ( l + h ) / 2 );
13        if(A[m] == k)      return true;
14        else if(A[m] < k) l = m + 1;
15        else                h = m - 1;
16    }
17    return false;
18 }
```

- The while-loop dominates the algorithm.
- For simplicity, let's assume the while-loop takes a time to be executed once.
- In the worst case, the item we are looking for is the first or the last item of the array, the while-loop will be executed $\log_2(N)$ times.

Example 3

- Given $T(n) = a\log_2(n)$, let's prove $T(n) \in O(\log_2(n))$.

Let $g(n) = \log_2(n)$, $k = 2a$, $n_0 = 1$,

obviously, $a\log_2(n) \leq 2a\log_2(n)$ when $n > 1$.

Hence, $T(n) \in O(\log_2(n))$.

Example 4

```
2 // A is an array of N integer items to be sorted
3 void bubbleSortNonDesending(std::vector<int> & A)
4 {
5     bool ff_not_sorted = true;
6     while(ff_not_sorted)
7     {
8         ff_not_sorted = false;
9         for(int i = 1; i < A.size(); ++i)
10        {
11            if(A[i-1] > A[i])
12            {
13                int temp = A[i-1];
14                A[i-1] = A[i];
15                A[i] = temp;
16                ff_not_sorted = true;
17            }
18        }
19    }
20 }
```

- The while-loop dominates the algorithm. The for-loop dominates the body of while-loop.
- Inside the for-loop, assume it takes a time to execute the if-statement if the condition is true, otherwise it takes b time to execute the if-statement.

Example 4 – an instance of worst case

```
2 // A is an array of N integer items to be sorted
3 void bubbleSortNonDesending(std::vector<int> & A)
4 {
5     bool ff_not_sorted = true;
6     while(ff_not_sorted)
7     {
8         ff_not_sorted = false;
9         for(int i = 1; i < A.size(); ++i)
10        {
11            if(A[i-1] > A[i])
12            {
13                int temp = A[i-1];
14                A[i-1] = A[i];
15                A[i] = temp;
16                ff_not_sorted = true;
17            }
18        }
19    }
20 }
```

- In the worst case, the input array has N items in ascending order.
- For instance,
$$A = \{5, 4, 3, 2, 1\}$$

Example 4 – an instance of worst case

```
2 // A is an array of N integer items to be sorted
3 void bubbleSortNonDesending(std::vector<int> & A)
4 {
5     bool ff_not_sorted = true;
6     while(ff_not_sorted)
7     {
8         ff_not_sorted = false;
9         for(int i = 1; i < A.size(); ++i)
10        {
11            if(A[i-1] > A[i])
12            {
13                int temp = A[i-1];
14                A[i-1] = A[i];
15                A[i] = temp;
16                ff_not_sorted = true;
17            }
18        }
19    }
20 }
```

- The 1st repeat of while-loop:

- Before for-loop:

$$A = \{5, 4, 3, 2, 1\}$$

$$i = 1:$$

$$A = \{4, 5, 3, 2, 1\}$$

$$i = 2:$$

$$A = \{4, 3, 5, 2, 1\}$$

$$i = 3:$$

$$A = \{4, 3, 2, 5, 1\}$$

$$i = 4:$$

$$A = \{4, 3, 2, 1, 5\}$$

Example 4 – an instance of worst case

- The *2nd* repeat of while-loop:
- Before for-loop:
 $A = \{4, 3, 2, 1, 5\}$
 $i = 1:$
 $A = \{3, 4, 2, 1, 5\}$
 $i = 2:$
 $A = \{3, 2, 4, 1, 5\}$
 $i = 3:$
 $A = \{3, 2, 1, 4, 5\}$
 $i = 4:$
 $A = \{3, 2, 1, 4, 5\}$
- The *3rd* repeat of while-loop:
- Before for-loop:
 $A = \{3, 2, 1, 4, 5\}$
 $i = 1:$
 $A = \{2, 3, 1, 4, 5\}$
 $i = 2:$
 $A = \{2, 1, 3, 4, 5\}$
 $i = 3:$
 $A = \{2, 1, 3, 4, 5\}$
 $i = 4:$
 $A = \{2, 1, 3, 4, 5\}$

Example 4 – an instance of worst case

- The 4th repeat of while-loop:
- Before for-loop:
 - $A = \{2, 1, 3, 4, 5\}$
 - $i = 1:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 2:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 3:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 4:$
 - $A = \{1, 2, 3, 4, 5\}$
- The 5th repeat of while-loop:
- Before for-loop:
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 1:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 2:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 3:$
 - $A = \{1, 2, 3, 4, 5\}$
 - $i = 4:$
 - $A = \{1, 2, 3, 4, 5\}$

Example 4 – an instance of worst case

```
2 // A is an array of N integer items to be sorted
3 void bubbleSortNonDesending(std::vector<int> & A)
4 {
5     bool ff_not_sorted = true;
6     while(ff_not_sorted)
7     {
8         ff_not_sorted = false;
9         for(int i = 1; i < A.size(); ++i)
10        {
11            if(A[i-1] > A[i])
12            {
13                int temp = A[i-1];
14                A[i-1] = A[i];
15                A[i] = temp;
16                ff_not_sorted = true;
17            }
18        }
19    }
20 }
```

- Since the 5th repeat of while-loop doesn't swap any item, the flag *ff_not_sorted* will be true, so the while-loop condition will be false, the algorithm will be terminated.

Example 4 – an instance of worst case

```
2 // A is an array of N integer items to be sorted
3 void bubbleSortNonDesending(std::vector<int> & A)
4 {
5     bool ff_not_sorted = true;
6     while(ff_not_sorted)
7     {
8         ff_not_sorted = false;
9         for(int i = 1; i < A.size(); ++i)
10        {
11            if(A[i-1] > A[i])
12            {
13                int temp = A[i-1];
14                A[i-1] = A[i];
15                A[i] = temp;
16                ff_not_sorted = true;
17            }
18        }
19    }
20 }
```

- In summary,

$$\begin{aligned} T(5) \\ &= (4a) + (3a + b) + (2a + 2b) \\ &\quad + (a + 3b) + (4b) \\ &= (a + b) \frac{4(1 + 4)}{2} \end{aligned}$$

- More generally,

$$T(n) = (a + b) \frac{n(n - 1)}{2}$$

Example 4

- Given $T(n) = (a + b) \frac{n(n-1)}{2}$, let's prove $T(n) \in O(n^2)$.

Let $g(n) = n^2$, $k = a + b$, $n_0 = 1$,

obviously, when $n > 1$,

$$\begin{aligned}n &\leq n^2, \\ \frac{(a+b)}{2} n &\leq \frac{(a+b)}{2} n^2 \\ \frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n &\leq \frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n^2 \\ T(n) = \frac{(a+b)}{2} n^2 - \frac{(a+b)}{2} n &< \frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n \leq (a+b)n^2\end{aligned}$$

Hence, $T(n) \in O(n^2)$.

Example 5 – Multiplication of matrices

```
1 import numpy as np
2
3 # A and B are both n-by-n matrices
4 def multipleMatrices(A, B):
5     if(A.shape[1] != B.shape[0]):
6         return None
7     ans = np.zeros((A.shape[0], B.shape[1]))
8     for i in range(A.shape[0]):
9         for j in range(B.shape[1]):
10            for k in range(A.shape[1]):
11                ans[i,j] += A[i,k]*B[k,j]
12
13 return ans
```

- The nested for-loop dominates the algorithm.
- For simplicity, assuming both matrices are n -by- n square matrices.
- Assume it takes a time to execute line 11.

- In summary,

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n a = an^3$$

Example 6 – merge sort

```
18 void mergeSort(vector<int>& A)
19 {
20     mergeSortSplit(A, 0, A.size());
21 }
22
23 void mergeSortSplit(vector<int>& A, int begin, int end)
24 {
25     if(end - begin <= 1) return; //one item itself is sorted
26     int middle = int((begin + end) / 2);
27     mergeSortSplit(A, begin, middle);
28     mergeSortSplit(A, middle, end);
29     mergeSortMerge(A, begin, middle, end);
30 }
31
32 void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
33 {
34     vector<int> B(A.begin() + begin, A.begin() + middle);
35     vector<int> C(A.begin() + middle, A.begin() + end);
36
37     int p = 0, q = 0, k = begin;
38     while(p < B.size() && q < C.size())
39     {
40         if(B[p] < C[q]) A[k++] = B[p++];
41         else             A[k++] = C[q++];
42     }
43     while(p < B.size()) A[k++] = B[p++];
44     while(q < C.size()) A[k++] = C[q++];
45 }
```

- Merge sort is a typical divide and conquer algorithm.

- During the divide stage, the original problem is divided into equally two half, so

$$T(N) = 2T\left(\frac{N}{2}\right) + f(N)$$

- $f(N)$ is the time taken to merge the subproblem (function *mergeSortMerge*).

Example 6 – merge sort

```
18 void mergeSort(vector<int>& A)
19 {
20     mergeSortSplit(A, 0, A.size());
21 }
22
23 void mergeSortSplit(vector<int>& A, int begin, int end)
24 {
25     if(end - begin <= 1) return; //one item itself is sorted
26     int middle = int((begin + end) / 2);
27     mergeSortSplit(A, begin, middle);
28     mergeSortSplit(A, middle, end);
29     mergeSortMerge(A, begin, middle, end);
30 }
31
32 void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
33 {
34     vector<int> B(A.begin() + begin, A.begin() + middle);
35     vector<int> C(A.begin() + middle, A.begin() + end);
36
37     int p = 0, q = 0, k = begin;
38     while(p < B.size() && q < C.size())
39     {
40         if(B[p] < C[q]) A[k++] = B[p++];
41         else             A[k++] = C[q++];
42     }
43     while(p < B.size()) A[k++] = B[p++];
44     while(q < C.size()) A[k++] = C[q++];
45 }
```

- In function *mergeSortMerge*, the array B and array C sorted the sorted sub-array and the 3 while-loop will merge the 2 sorted subarray into 1 array.
- In each repeat of any of the while-loop, one element is put into where it's supposed to be.

Example 6 – merge sort

```
18 void mergeSort(vector<int>& A)
19 {
20     mergeSortSplit(A, 0, A.size());
21 }
22
23 void mergeSortSplit(vector<int>& A, int begin, int end)
24 {
25     if(end - begin <= 1) return; //one item itself is sorted
26     int middle = int((begin + end) / 2);
27     mergeSortSplit(A, begin, middle);
28     mergeSortSplit(A, middle, end);
29     mergeSortMerge(A, begin, middle, end);
30 }
31
32 void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
33 {
34     vector<int> B(A.begin() + begin, A.begin() + middle);
35     vector<int> C(A.begin() + middle, A.begin() + end);
36
37     int p = 0, q = 0, k = begin;
38     while(p < B.size() && q < C.size())
39     {
40         if(B[p] < C[q]) A[k++] = B[p++];
41         else             A[k++] = C[q++];
42     }
43     while(p < B.size()) A[k++] = B[p++];
44     while(q < C.size()) A[k++] = C[q++];
45 }
```

- The 3 while-loops are dominant. Assume it takes a time to execute one repeat of the while-loop.

$$f(N) = aN$$

- Where, N is the total sizes of the 2 subproblems. So,

$$T(N) = 2T\left(\frac{N}{2}\right) + aN$$

- Solve it,

$$T(N) = N + a \cdot N \cdot \log_2(N)$$

$$T(N) \in O(N \log_2(N))$$

Example 6 – merge sort

$$T(N) = 2T\left(\frac{N}{2}\right) + aN, \text{ so, } T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{2^2}\right) + a\frac{N}{2}, T\left(\frac{N}{2^2}\right) = 2T\left(\frac{N}{2^3}\right) + a\frac{N}{2^2}$$

Plugging in, we have

$$T(N) = 2\left(2T\left(\frac{N}{2^2}\right) + a\frac{N}{2}\right) + aN = 2^2T\left(\frac{N}{2^2}\right) + aN + aN$$

$$T(N) = 2^2\left(2T\left(\frac{N}{2^3}\right) + a\frac{N}{2^2}\right) + aN + aN = 2^3T\left(\frac{N}{2^3}\right) + aN + aN + aN$$

..., assuming $N = 2^k$

$$T(N) = 2^kT\left(\frac{N}{2^k}\right) + akN = N + aN\log_2(N)$$

Example 7 – compute Fibonacci 1

```
2 // naive computation of fibonacci
3 void fibonacci(int N)
4 {
5     int ans = 0
6     if(N == 0)      return 0;
7     else if(N < 2)  return 1;
8     int f1 = 0, f2 = 1;
9     for(int i = 2; i <= N; ++i)
10    {
11        int ans = f1 + f2;
12        f1 = f2;
13        f2 = ans;
14    }
15    return ans;
16 }
```

- The for-loop dominate the algorithm.
- Assume it takes a time to execute the body of for-loop.
- $T(N) = a(N - 1) \in O(N)$.

Example 8 – compute Fibonacci 2

```
2 // naive computation of fibonacci
3 void fibonacci(int N)
4 {
5     if(N == 0)    return 0;
6     if(N == 1)    return 1;
7     return fibonacci(N-1)+fibonacci(N-2);
8 }
```

- The running time of the algorithm:

$$T(N) = T(N - 1) + T(N - 2) + 1$$

- Solving the linear recursive equation:

$$T(N) = \left(\frac{1 + \sqrt{5}}{2}\right)^N + \left(\frac{1 - \sqrt{5}}{2}\right)^N$$

Example 8

- Given $T(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$, let's prove $T(n) \in O(2^n)$.

Let $g(n) = 2^n$, $k = 2$, $n_0 = 1$,

obviously, when $n > 1$,

$$\frac{1 + \sqrt{5}}{2} \leq 2 \rightarrow \left(\frac{1 + \sqrt{5}}{2}\right)^n \leq 2^n$$

$$\frac{1 - \sqrt{5}}{2} \leq 2 \rightarrow \left(\frac{1 - \sqrt{5}}{2}\right)^n \leq 2^n$$

$$T(n) \leq 2 \cdot 2^n$$

Hence, $T(n) \in O(2^n)$.