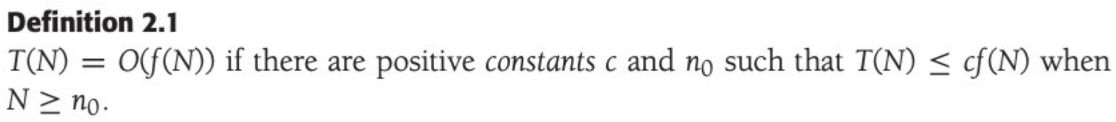
Algorithm Analysis



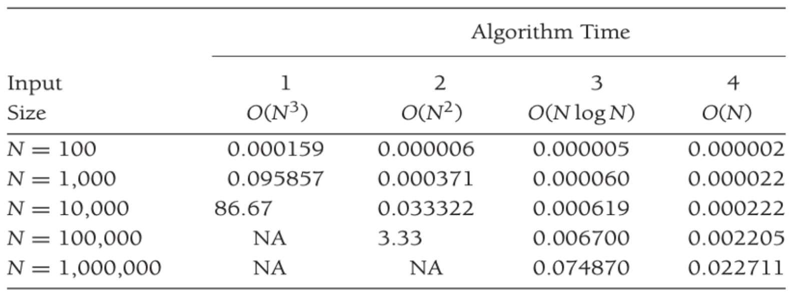
Definition 2.1

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

O(f(N)) indicates the worst time or upper bound to total time for a algorithm
T(N) or time for processing input size of “N”

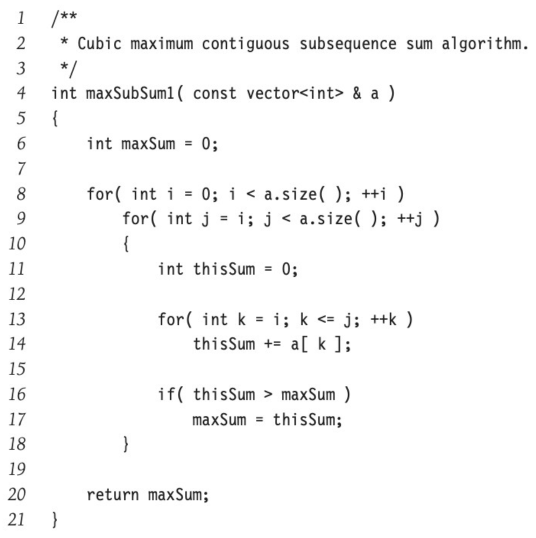
MERGE SORT, TWO POINTER, SLIDING WINDOW, [redacted]

Maximum Subsequence Problem



Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

$O(N^3)$



```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Figure 2.5 Algorithm 1

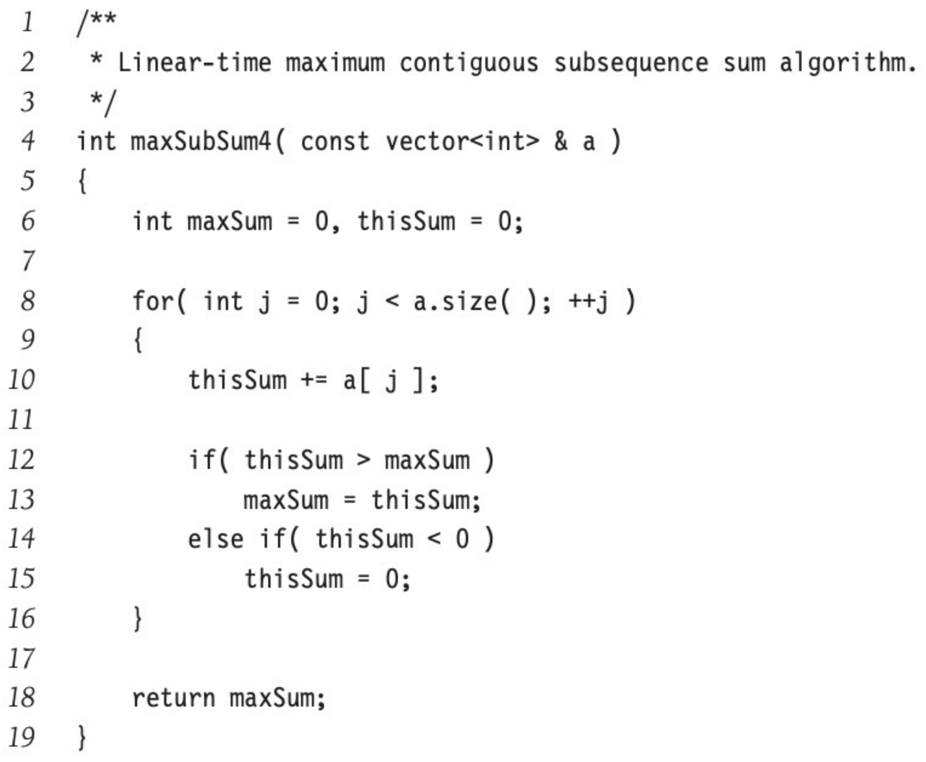
$O(N^2)$

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

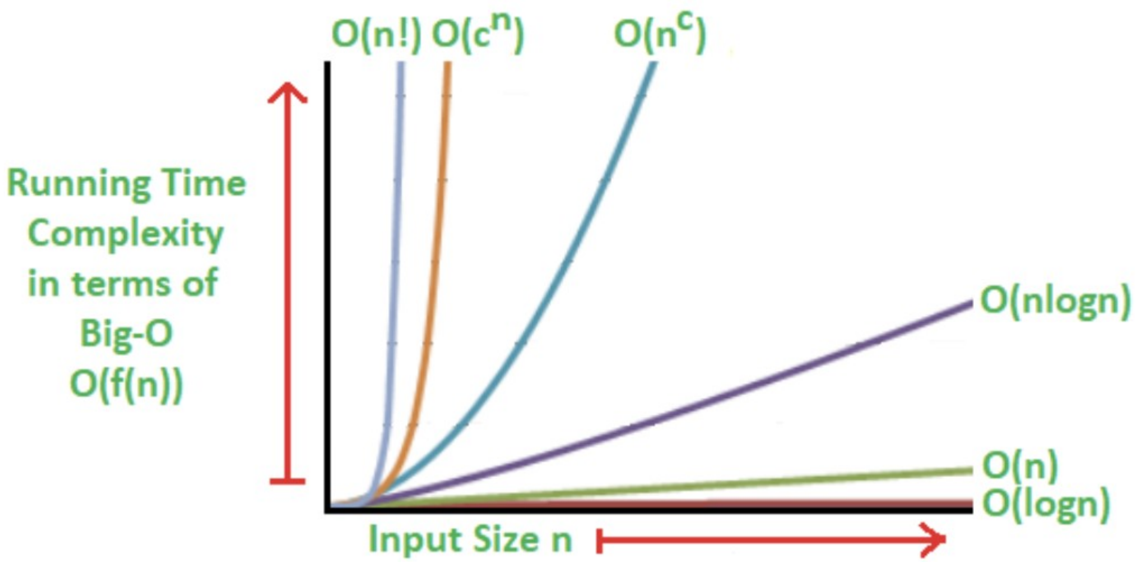
O(N log(N))

```
/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base case
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center = ( left + right ) / 2;
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for( int i = center; i >= left; --i )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    int maxRightBorderSum = 0, rightBorderSum = 0;
    for( int j = center + 1; j <= right; ++j )
    {
        rightBorderSum += a[ j ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                    maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Driver for divide-and-conquer maximum contiguous
 * subsequence sum algorithm.
 */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

O(N)



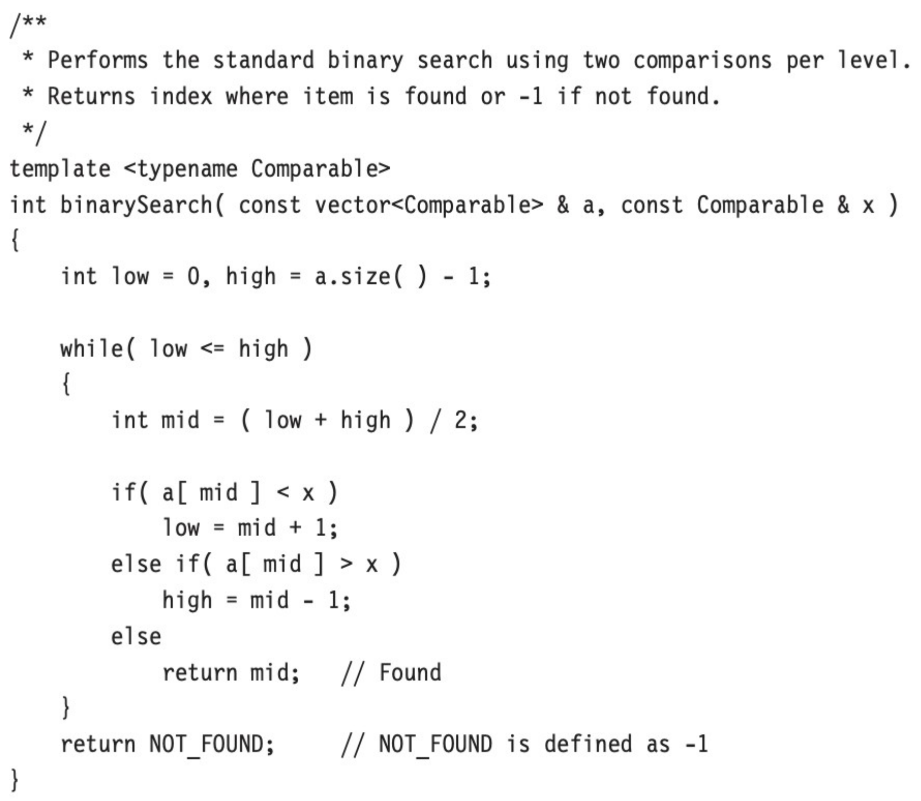
```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); ++j )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```



following general rule: *An algorithm is O*(log *N*) *if it takes constant* (*O*(1)) *time to cut the problem size by a fraction (which is usually* $\frac{1}{2}$ *).* On the other hand, if constant time is required to merely reduce the problem by a constant *amount* (such as to make the problem smaller by 1), then the algorithm is *O*(*N*).

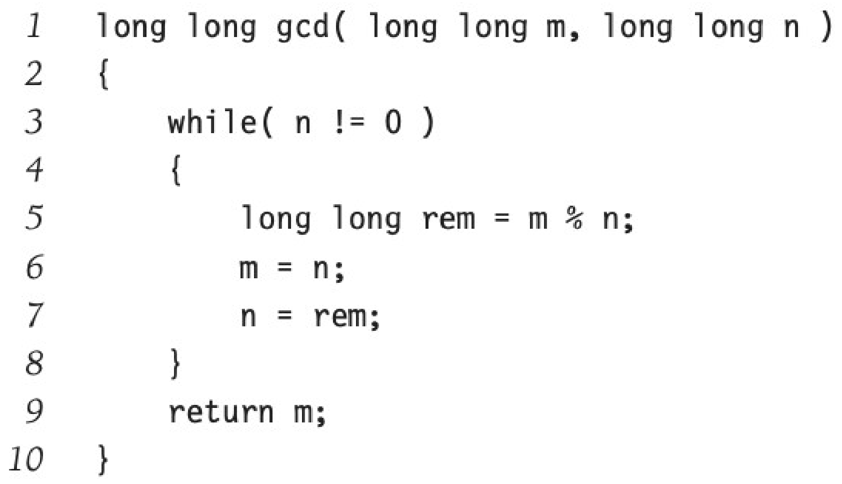
since we take best case (N) just to read all the elements, to create a algorithm with log(N) we would need a sorted data structure where reading it is not necessary like **binary search** and **euclid algorithm**

BINARY SEARCH (Olog(N))



```
/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found.
 */
template <typename Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid;   // Found
    }
    return NOT_FOUND;     // NOT_FOUND is defined as -1
}
```

EUCLID ALGORITHM (O(log (minimum of m and n))



```
long long gcd( long long m, long long n )
{
    while( n != 0 )
    {
        long long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

TwoSum on a Sorted Array using TwoPointer Approach

Copy code

```cpp
#include <iostream>
#include <vector>

std::vector<int> twoSum(const std::vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;

    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return {left, right};
        } else if (sum < target) {
            ++left;
        } else {
            --right;
        }
    }

    return {};
}
```

since each element is traversed once, time complexity is O(N)

TwoPointers are good for finding one pair of values in a array that satisfy constaints

Sliding Window Approach using Kardane's Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

int maxSubarraySum(vector<int> &arr) {
    int currentSum = arr[0];
    int maxSum = arr[0];

    for (int i = 1; i < arr.size(); i++) {
        currentSum = max(currentSum + arr[i], arr[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}

int main() {
    vector<int> arr = {2, 3, -8, 7, -1, 2, 3};
    cout << maxSubarraySum(arr);
    return 0;
}
```

since each element is traversed once, time complexity is O(N)

SlidingWindow are good for finding a subarray that satisfies the constraints

LinkedList vs Arrays/Vectors

Advantages of LinkedList

1) while LinkedList and vectors don’t have any fixed size, arrays have size limit on them.

2)while Vectors are dynamic, their resizing is costly as it often involves copying the whole vector to a new memory location.

3) insertion/deletion in arrays and vectors take O(N) time due to shifting of all elements but in LinkedList takes O(1) time.

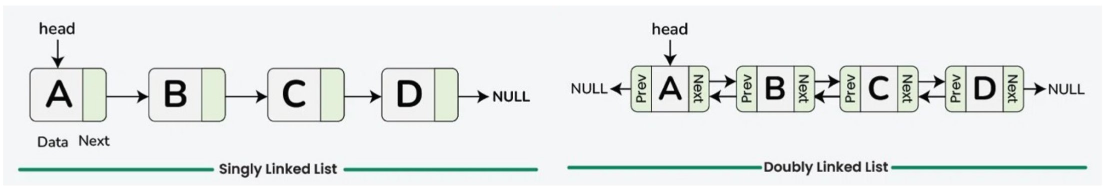
4) the memory doesn’t have to available in sequential blocks

Disadvantages of LinkedList

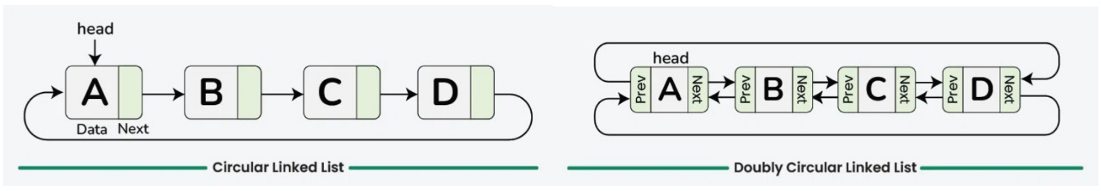
1)Arrays and Vectors are generally more memory efficient due to contiguous storage of elements

2) Arrays/Vectors can randomly accessed in O(1) time while LinkedList usually don’t have indexes and are traversed via running pointers

3) they use cpu cache better and hence better performance



doubly-linked list supports traversal in both forward and backward directions and makes deletions easy as u can directly access previous and next nodes.



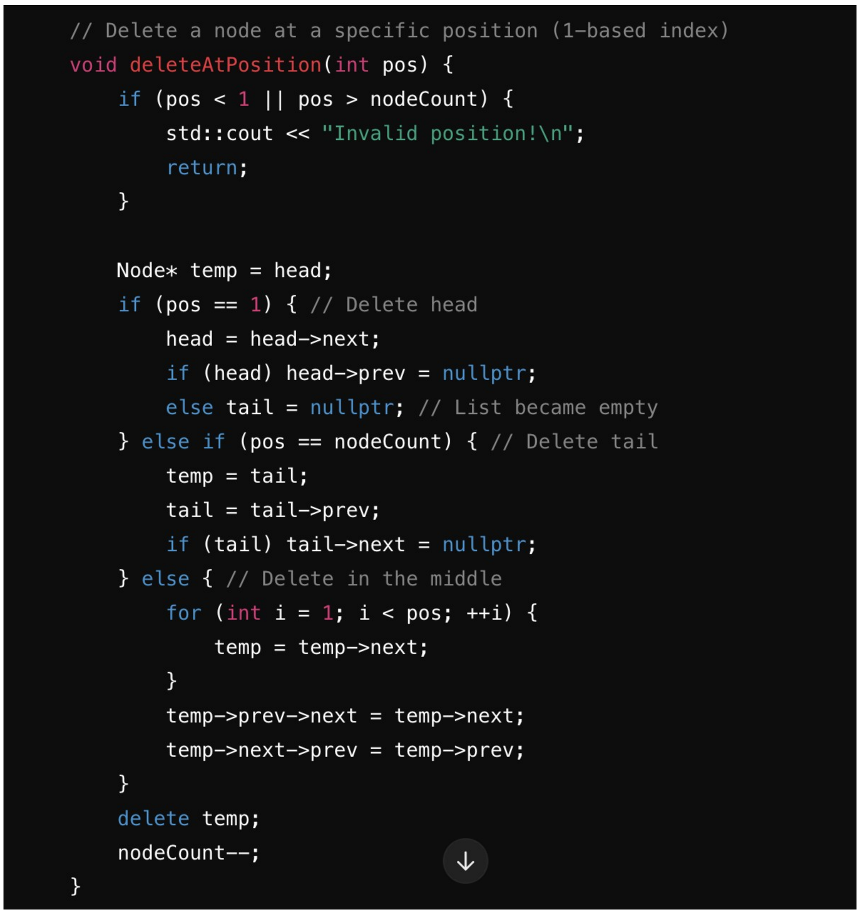
circular linked-list helps with repeated tasks like cpu scheduling or playlist managers of song applications

LinkedList in C++ using User-Defined Type

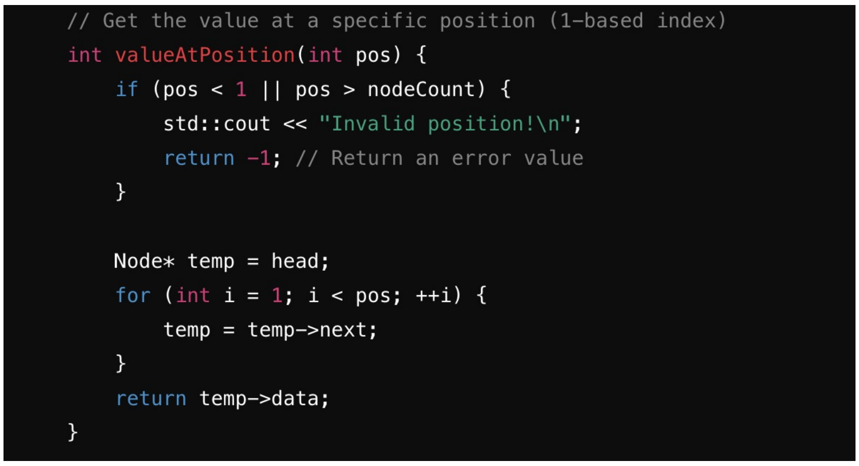
```cpp
#include <iostream>

struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

```cpp
class LinkedList {
private:
    Node* head;
    Node* tail;
    int nodeCount; // keeps track of the number of nodes
```

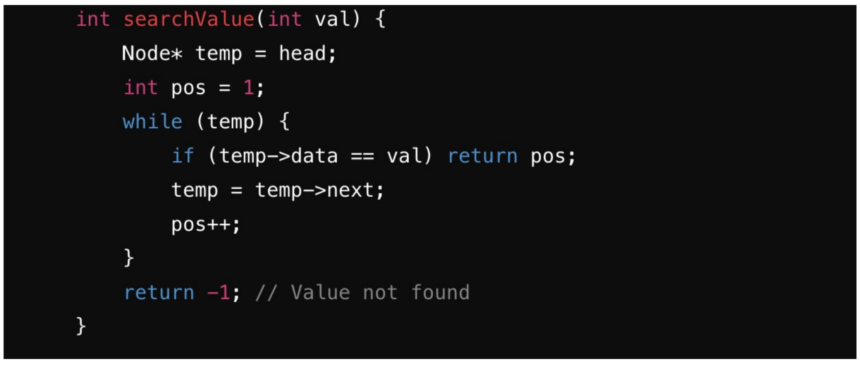
```cpp
public:
    LinkedList() : head(nullptr), tail(nullptr), nodeCount(0) {}

    // Insert a node at a specific position (1-based index)
    void insertAtPosition(int pos, int val) {
        if (pos < 1 || pos > nodeCount + 1) {
            std::cout << "Invalid position!\n";
            return;
        }

        Node* newNode = new Node(val);
        if (pos == 1) { // Insert at head
            newNode->next = head;
            if (head) head->prev = newNode;
            head = newNode;
            if (!tail) tail = newNode; // If list was empty
        } else if (pos == nodeCount + 1) { // Insert at tail
            newNode->prev = tail;
            if (tail) tail->next = newNode;
            tail = newNode;
        } else { // Insert in the middle
            Node* temp = head;
            for (int i = 1; i < pos - 1; ++i) {
                temp = temp->next;
            }
            newNode->next = temp->next;
            newNode->prev = temp;
            temp->next->prev = newNode;
            temp->next = newNode;
        }
        nodeCount++;
    }
```



```cpp
// Delete a node at a specific position (1-based index)
void deleteAtPosition(int pos) {
    if (pos < 1 || pos > nodeCount) {
        std::cout << "Invalid position!\n";
        return;
    }

    Node* temp = head;
    if (pos == 1) { // Delete head
        head = head->next;
        if (head) head->prev = nullptr;
        else tail = nullptr; // List became empty
    } else if (pos == nodeCount) { // Delete tail
        temp = tail;
        tail = tail->prev;
        if (tail) tail->next = nullptr;
    } else { // Delete in the middle
        for (int i = 1; i < pos; ++i) {
            temp = temp->next;
        }
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }
    delete temp;
    nodeCount--;
}
```



```cpp
// Get the value at a specific position (1-based index)
int valueAtPosition(int pos) {
    if (pos < 1 || pos > nodeCount) {
        std::cout << "Invalid position!\n";
        return -1; // Return an error value
    }

    Node* temp = head;
    for (int i = 1; i < pos; ++i) {
        temp = temp->next;
    }
    return temp->data;
}
```



```cpp
int searchValue(int val) {
    Node* temp = head;
    int pos = 1;
    while (temp) {
        if (temp->data == val) return pos;
        temp = temp->next;
        pos++;
    }
    return -1; // Value not found
}
```

```
    // Get the number of nodes in the list
    int getNodeCount() const {
        return nodeCount;
    }

    // Display the list
    void displayList() const {
        Node* temp = head;
        while (temp) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << "\n";
    }
};
```

```
int main() {
    LinkedList list;

    // Insert nodes
    list.insertAtPosition(1, 10);
    list.insertAtPosition(2, 20);
    list.insertAtPosition(3, 30);
    list.insertAtPosition(2, 15); // Insert 15 at position 2

    std::cout << "List after insertion: ";
    list.displayList(); // Output: 10 15 20 30

    // Delete a node
    list.deleteAtPosition(3); // Delete the node at position 3
    std::cout << "List after deletion at position 3: ";
    list.displayList(); // Output: 10 15 30
```

```
    // Search for a value
    int val = 30;
    int foundPos = list.searchValue(val);
    if (foundPos != -1) {
        std::cout << "Value " << val << " found at position " << foundPos << "\n";
    } else {
        std::cout << "Value " << val << " not found in the list.\n";
    }
```

```
    // Get the number of nodes
    std::cout << "Number of nodes: " << list.getNodeCount() << "\n";

    return 0;
}
```

Class Implementations

in C++, there is a library of class implementation of common data structures called STL (Standard Template Library). It has containers like list(doubly linked list), vectors(dynamic arrays), heaps, trees and hash tables

we have some methods for these class templates like

std::vector<int> vectorname;

vectorname.push_back(10);

vectorname.pop_back();

vectorname.front();

vectorname.back();

std::list<int> listname;

listname.push_back(10);

listname.pop_back();

listname.front();

listname.back();

//only available for doubly linkedlist due to better pushing and popping at front of container

listname.push_front(10);

listname.pop_front();

Iterators

In C++, **iterators** are objects that provide a standardized way to access elements within STL containers (like std::vector, std::list, std::map, etc.). Iterators act like pointers, allowing you to traverse or manipulate elements in containers sequentially without exposing the container's underlying implementation details.

Iterators behave like pointers in sense that operators like deference (*****), increment/decrement(**++/--**) and equality checker (**==/!=**)

Iterators have different datatype depending on container its pointing towards like **std::vector<int>::iterator iteratorname;** or **std::list<int>::iterator iteratorname;**

we can also use auto to initialise the iterator with some functions that return iterator datatypes

for (auto it = vectorname.begin(); it != vectorname.end(); ++it)

we also have other functions

listname.insert(iterator position,value);

listname.erase(iterator position);

listname.erase(listname.begin(),listname.end());

In C++, a **const iterator** is a type of iterator that allows you to traverse a container without modifying its elements. Const iterators are useful when you need to read data without risking unintended changes to the container's elements.

std::vector<int>::const_iterator name;

cbegin() and **cend()** return const_iterators

Implementation of Vector in Class

```
#include <algorithm>

template <typename Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 ) : theSize{ initSize },
         theCapacity{ initSize + SPARE_CAPACITY }
      { objects = new Object[ theCapacity ]; }

    Vector( const Vector & rhs ) : theSize{ rhs.theSize },
        theCapacity{ rhs.theCapacity }, objects{ nullptr }
    {
        objects = new Object[ theCapacity  ];
        for( int k = 0; k < theSize; ++k )
            objects[ k ] = rhs.objects[ k ];
    }

    Vector & operator= ( const Vector & rhs )
    {
        Vector copy = rhs;
        std::swap( *this, copy );
        return *this;
    }

    ~Vector( )
      { delete [ ] objects; }

    Vector( Vector && rhs ) : theSize{ rhs.theSize },
        theCapacity{ rhs.theCapacity }, objects{ rhs.objects }
    {
        rhs.objects = nullptr;
        rhs.theSize = 0;
        rhs.theCapacity = 0;
    }

    Vector & operator= ( Vector && rhs )
    {
        std::swap( theSize, rhs.theSize );
        std::swap( theCapacity, rhs.theCapacity );
        std::swap( objects, rhs.objects );

        return *this;
    }

    void resize( int newSize )
    {
        if( newSize > theCapacity )
            reserve( newSize * 2 );
        theSize = newSize;
    }

    void reserve( int newCapacity )
    {
        if( newCapacity < theSize )
            return;

        Object *newArray = new Object[ newCapacity ];
        for( int k = 0; k < theSize; ++k )
            newArray[ k ] = std::move( objects[ k ] );

        theCapacity = newCapacity;
        std::swap( objects, newArray );
        delete [ ] newArray;
    }


    Object & operator[]( int index )
      { return objects[ index ]; }
    const Object & operator[]( int index ) const
      { return objects[ index ]; }

    bool empty( ) const
      { return size( ) == 0; }
    int size( ) const
      { return theSize; }
    int capacity( ) const
      { return theCapacity; }

    void push_back( const Object & x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    void push_back( Object && x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = std::move( x );
    }

    void pop_back( )
    {
        --theSize;
    }

    const Object & back ( ) const
    {
         return objects[ theSize - 1 ];
    }

    typedef Object * iterator;
    typedef const Object * const_iterator;

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

1. Like its STL counterpart, there is

limited error checking. Later we will briefly discuss how error checking can be provided.

As shown on lines 118 to 120, the Vector stores the size, capacity, and primitive array as its data members. The constructor at lines 7 to 9 allows the user to specify an initial size, which defaults to zero. It then initializes the data members, with the capacity slightly larger than the size, so a few push_backs can be performed without changing the capacity.

The copy constructor, shown at lines 11 to 17, makes a new Vector and can then be used by a casual implementation of operator= that uses the standard idiom of swapping in a copy. This idiom works only if swapping is done by moving, which itself requires the implementation of the move constructor and move operator= shown at lines 29 to 44. Again, these use very standard idioms. Implementation of the copy assignment operator= using a copy constructor and swap, while simple, is certainly not the most efficient method, especially in the case where both Vectors have the same size. In that special case, which can be tested for, it can be more efficient to simply copy each element one by one using Object's operator=.

The resize routine is shown at lines 46 to 51. The code simply sets the theSize data member, after possibly expanding the capacity. Expanding capacity is very expen- sive. So if the capacity is expanded, it is made twice as large as the size to avoid having to change the capacity again unless the size increases dramatically (the +1 is used in case the size is 0). Expanding capacity is done by the reserve routine, shown at lines 53 to 65. It consists of allocation of a new array at line 58, moving the old contents at lines 59 and 60, and the reclaiming of the old array at line 64. As shown at lines 55 and 56, the reserve routine can also be used to shrink the underlying array, but only if the specified new capacity is at least as large as the size. If it isn't, the reserve request is ignored.

The two versions of operator[] are trivial (and in fact very similar to the implementa- tions of operator[] in the matrix class in Section 1.7.2) and are shown in lines 67 to 70. Error checking is easily added by making sure that index is in the range 0 to size()-1, inclusive, and throwing an exception if it is not.

A host of short routines, namely, empty, size, capacity, push_back, pop_back, and back, are implemented in lines 72 to 101. At lines 83 and 90, we see the use of the postfix ++ operator, which uses theSize to index the array and then increases theSize. We saw the same idiom when discussing iterators: *itr++ uses itr to decide which item to view and then advances itr. The positioning of the ++ matters: In the prefix ++ operator, *++itr advances itr and then uses the new itr to decide which item to view, and likewise, objects[++theSize] would increment theSize and use the new value to index the array (which is not what we would want). pop_back and back could both benefit from error checks in which an exception is thrown if the size is 0.

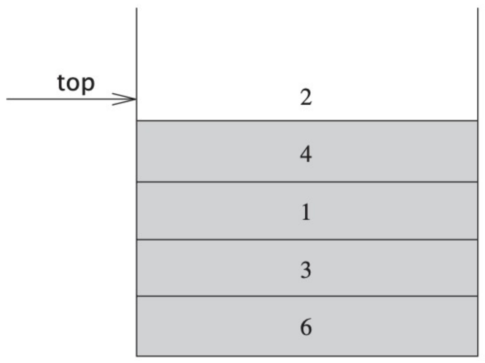
Finally, at lines 103 to 113 we see the declaration of the iterator and const_iterator nested types and the two begin and two end methods. This code makes use of the fact that in C++, a pointer variable has all the same operators that we expect for an iterator. Pointer variables can be copied and compared; the * operator yields the object being pointed at, and, most peculiarly, when ++ is applied to a pointer variable, the pointer variable then points at the object that would be stored next sequentially: If the pointer is pointing inside an array, incrementing the pointer positions it at the next array element. These semantics for pointers date back to the early 70s with the C programming language, upon which C++ is based. The STL iterator mechanism was designed in part to mimic pointer operations.

Consequently, at lines 103 and 104, we see typedef statements that state the iterator and const_iterator are simply other names for a pointer variable, and begin and end need to simply return the memory addresses representing the first array position and the first invalid array position, respectively.

The correspondence between iterators and pointers for the vector type means that using a vector instead of the C++ array is likely to carry little overhead. The disadvantage is that, as written, the code has no error checks. If the iterator itr goes crashing past the end marker, neither ++itr nor *itr will necessarily signal an error. To fix this problem would require that the iterator and const_iterator be actual nested class types rather than simply pointer variables.

Stacks

Stack is a list with restrictions that insertion and deletion can only be done at one end namely top of the stack. it has three functions **push,pop and top**(returns last inserted value)



Since a stack is a list, any list implementation will do. Clearly list and vector support stack operations; 99% of the time they are the most reasonable choice than using std::stack. Occasionally it can be faster to design a special-purpose implementation. Because stack operations are constant- time operations, this is unlikely to yield any discernable improvement except under very unique circumstances.

Application of Stacks

1) Stacks are used to check if there are balancing errors with braces and parenthesis in statements.

Make an empty stack. Read characters until end of file. If the character is an opening symbol, push it onto the stack. If it is a closing symbol and the stack is empty, report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At end of file, if the stack is not empty, report an error.

2) Stacks are used to find the solutions to arithmetic statements. Normal or Infix expressions can be converted into Postfix using Stack and then those Postfix expressions can be solved by computers without explicitly knowing precedence rules as operands are already in an order to provide clarity to the machine. Prefix expressions are not as widely used but also can be solved without knowledge of precedence rules.

3)Function calls are made using stacks. The current local variables in the function in order and return address(where to return once function is completed) are stored in a stack called **activation record** or **stack frame.** Each function call gets its own stack frame avoiding memory overwrite in recursion. This can also identify errors in form of stack overflow if few too many function calls are made without any end to the loop.

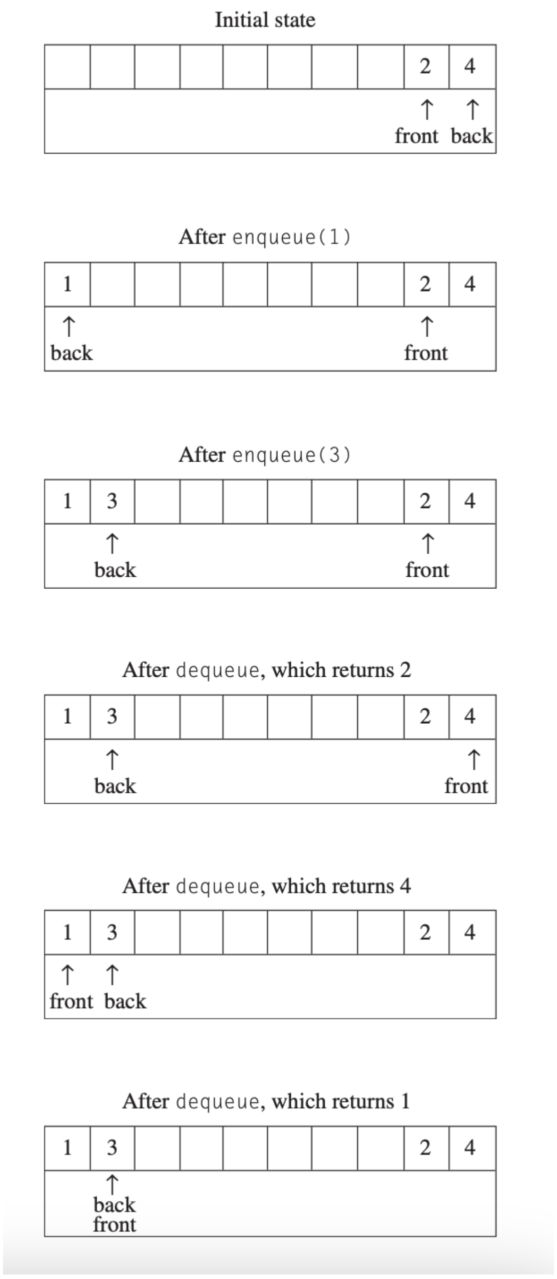
4) The undo option in text editors & back and forward options in browser viewing work on stack by popping the current state and restoring the previous state

5) Backtracking & DFS Algorithms that explore many different paths also use stacks

Array Implementation of Queues

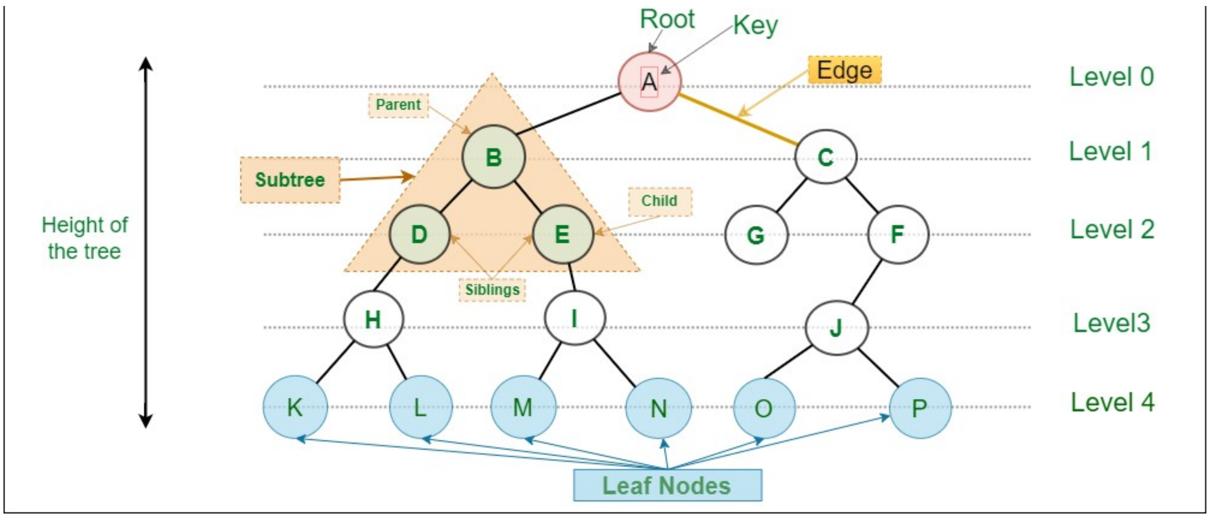
Like stacks, queues are also lists where insertion is done at the back end and the front end of the queue can be popped. The basic operations are enqueue, which inserts an element at the end of the list and dequeue that remove and returns the element at the front of the list.

Since a array is fixed in size, often times when we have dequeued some elements already, there is lots of free space in the beginning of the array and less space to enqueue new elements at the end of the array, making the array seem full when there is memory left. we use a circular array implementation in this case to work around this issue by wrapping the ends of array.


Initial state
2
4
front
back
After enqueue(1)
1
back
front
After enqueue(3)
3
After dequeue, which returns 2
After dequeue, which returns 4
After dequeue, which returns 1

Queues are used in data structures like graphs to give efficient running times but In simple daily life, they are used to manage first come-first serve problems like a printer printing in file order

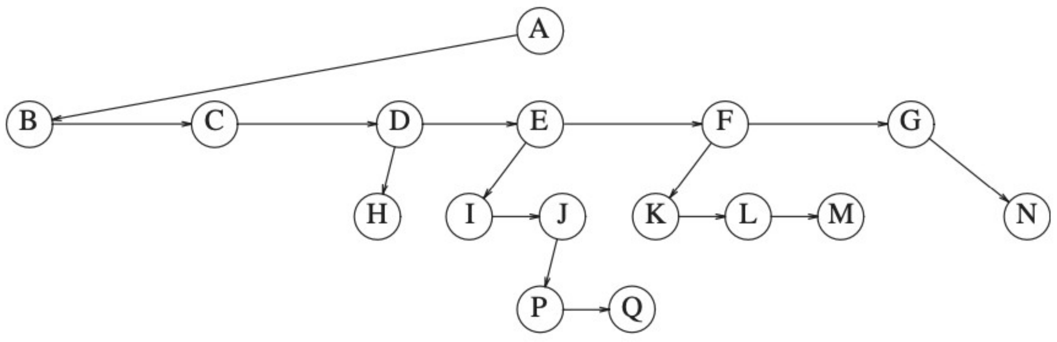
Trees



Trees are graphs with N nodes and N-1 edges with no cycles. The node with no parent is called **root node** and the level from root node is called **depth.** Nodes with no children are called **leaf nodes** and each node has one parent. Level from leaf node to a node is its **height.**

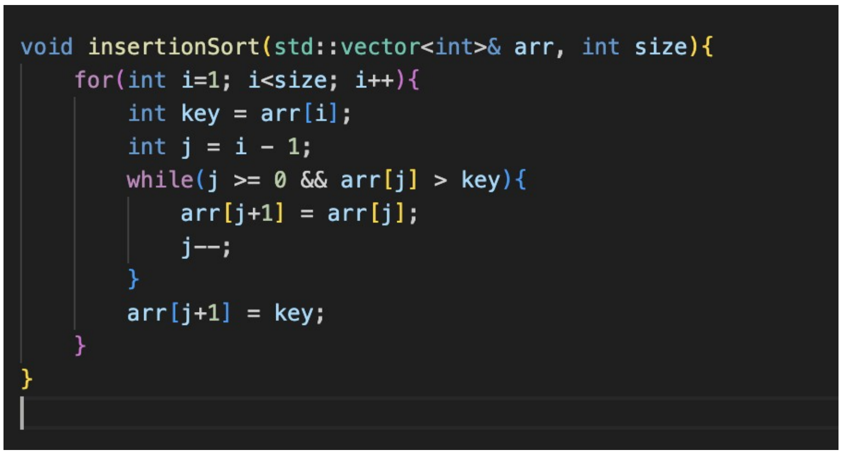
Node Representation

```
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
};
```



Sorting Methods

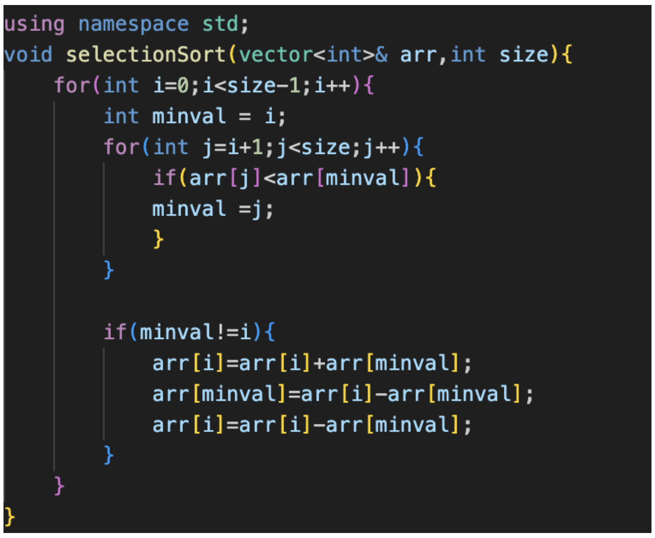
Insertion Sort



```cpp
void insertionSort(std::vector<int>& arr, int size){
    for(int i=1; i<size; i++){
        int key = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > key){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

Small arrays or nearly sorted arrays, often used in hybrid algorithms for tiny portions of data.

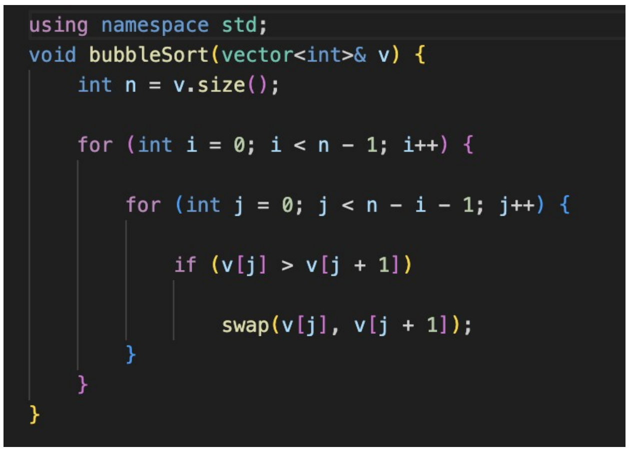
Selection Sort



```cpp
using namespace std;
void selectionSort(vector<int>& arr,int size){
    for(int i=0;i<size-1;i++){
        int minval = i;
        for(int j=i+1;j<size;j++){
            if(arr[j]<arr[minval]){
            minval =j;
            }
        }

        if(minval!=i){
            arr[i]=arr[i]+arr[minval];
            arr[minval]=arr[i]-arr[minval];
            arr[i]=arr[i]-arr[minval];
        }
    }
}
```

Simple and easy to implement for small lists but generally inefficient for large datasets.

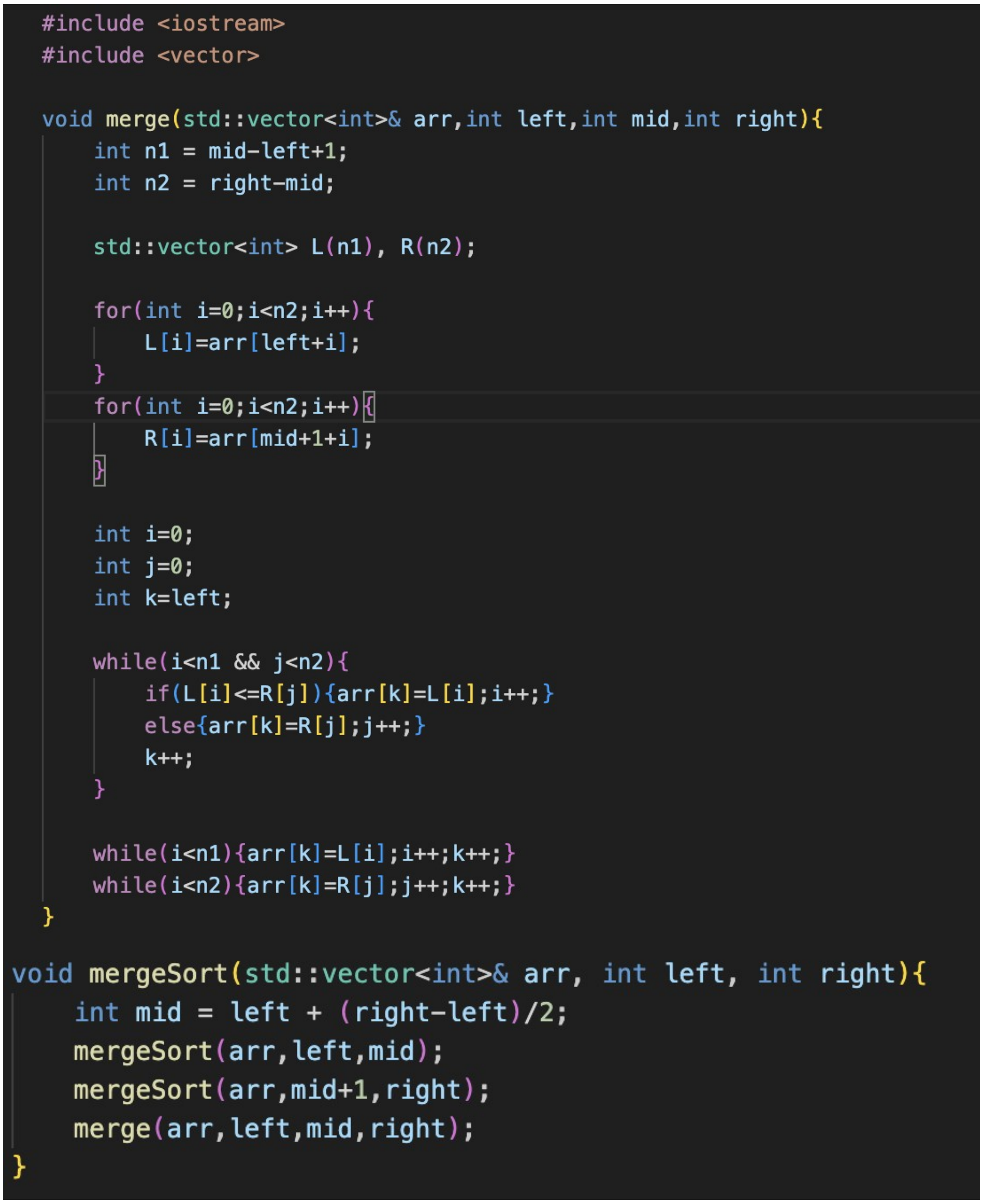
Bubble Sort



```cpp
using namespace std;
void bubbleSort(vector<int>& v) {
    int n = v.size();

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (v[j] > v[j + 1])

                swap(v[j], v[j + 1]);
        }
    }
}
```

Educational purposes or when sorting nearly sorted lists, though it is rarely used in practice.

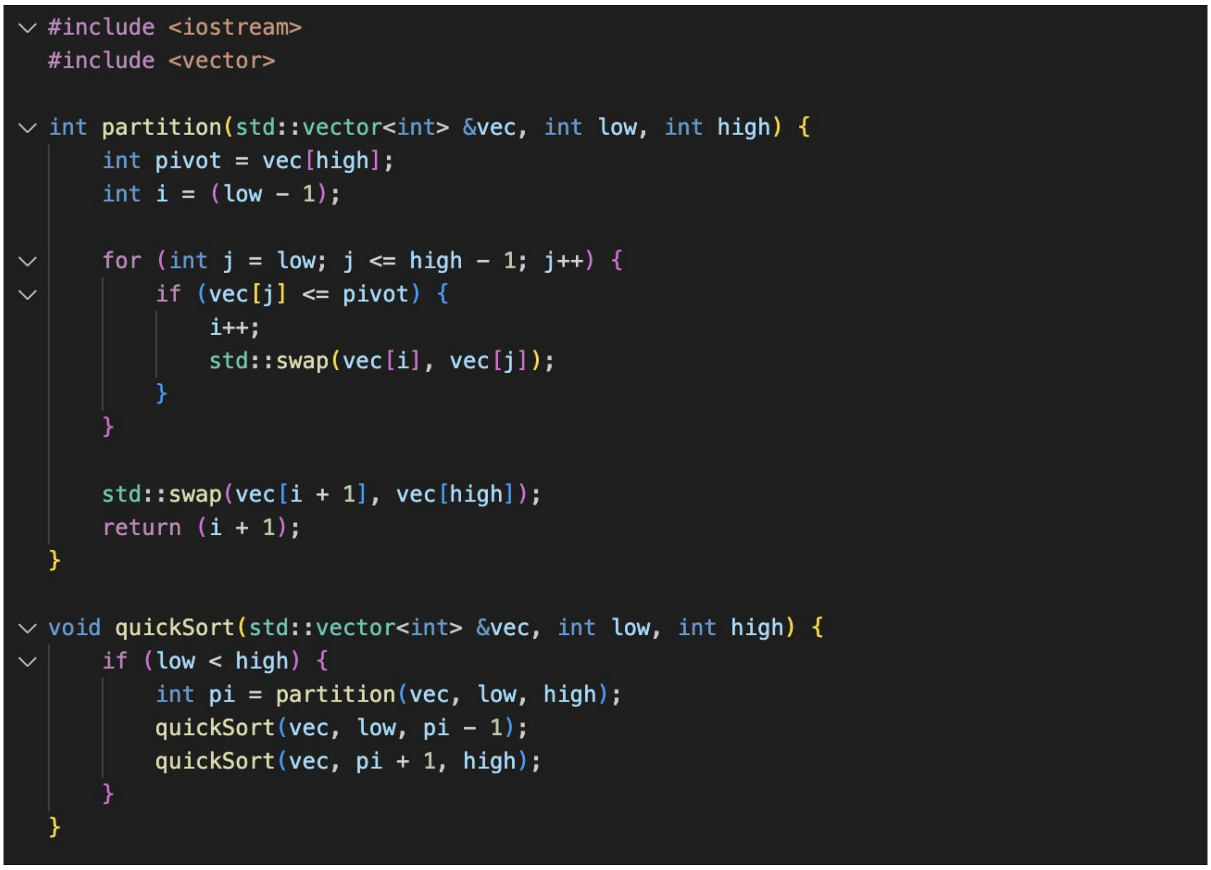
Merge Sort



```cpp
#include <iostream>
#include <vector>

void merge(std::vector<int>& arr,int left,int mid,int right){
    int n1 = mid-left+1;
    int n2 = right-mid;

    std::vector<int> L(n1), R(n2);

    for(int i=0;i<n2;i++){
        L[i]=arr[left+i];
    }
    for(int i=0;i<n2;i++){
        R[i]=arr[mid+1+i];
    }

    int i=0;
    int j=0;
    int k=left;

    while(i<n1 && j<n2){
        if(L[i]<=R[j]){arr[k]=L[i];i++;}
        else{arr[k]=R[j];j++;}
        k++;
    }

    while(i<n1){arr[k]=L[i];i++;k++;}
    while(i<n2){arr[k]=R[j];j++;k++;}
}

void mergeSort(std::vector<int>& arr, int left, int right){
    int mid = left + (right-left)/2;
    mergeSort(arr,left,mid);
    mergeSort(arr,mid+1,right);
    merge(arr,left,mid,right);
}
```

Large datasets where stability is important. Often used for external sorting on large files.

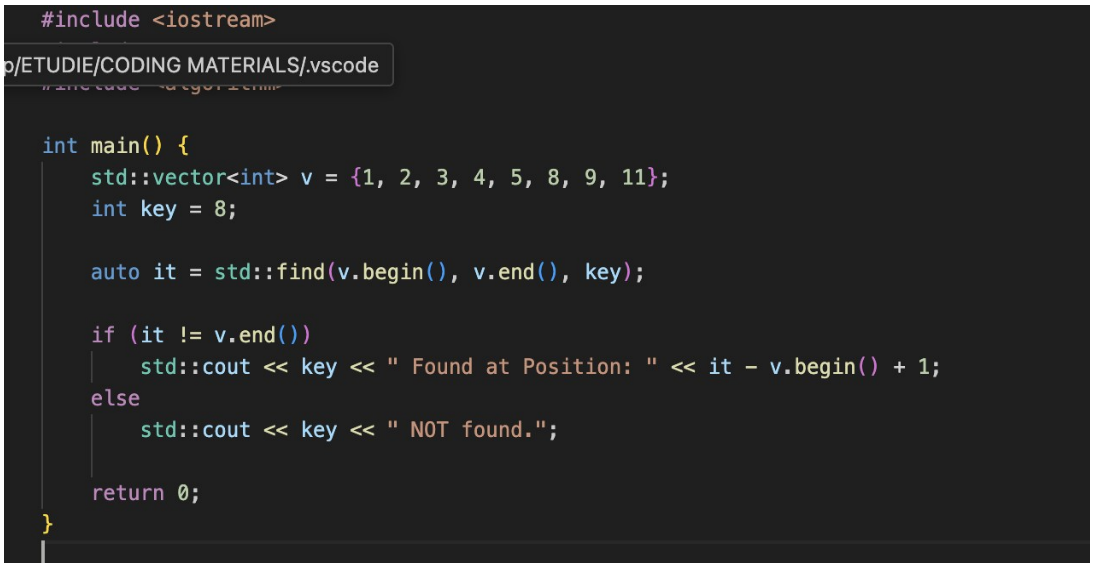
Quick Sort



```cpp
#include <iostream>
#include <vector>

int partition(std::vector<int> &vec, int low, int high) {
    int pivot = vec[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (vec[j] <= pivot) {
            i++;
            std::swap(vec[i], vec[j]);
        }
    }

    std::swap(vec[i + 1], vec[high]);
    return (i + 1);
}

void quickSort(std::vector<int> &vec, int low, int high) {
    if (low < high) {
        int pi = partition(vec, low, high);
        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}
```

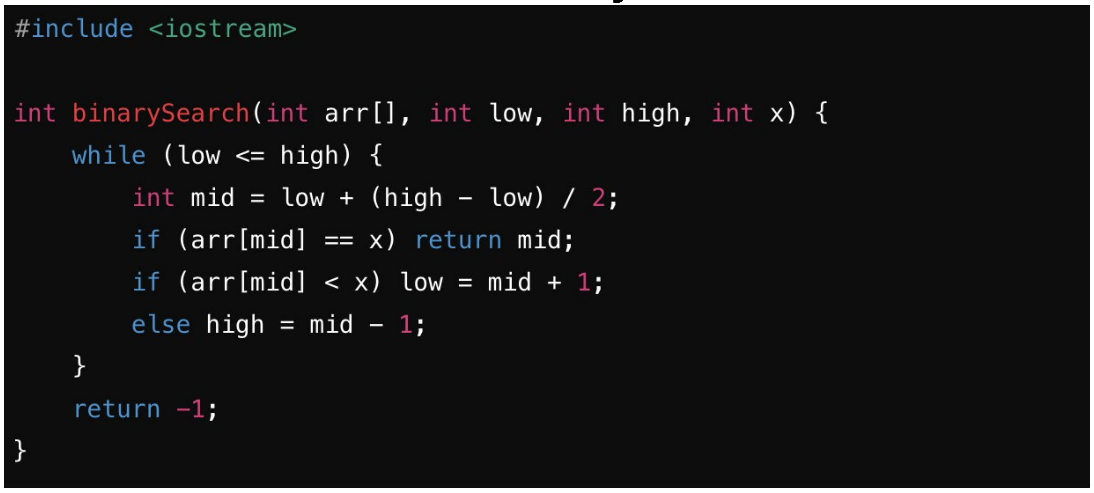
Large datasets, particularly when memory usage should be minimized. Often optimized with random pivots or hybridized for small partitions.

Linear Search



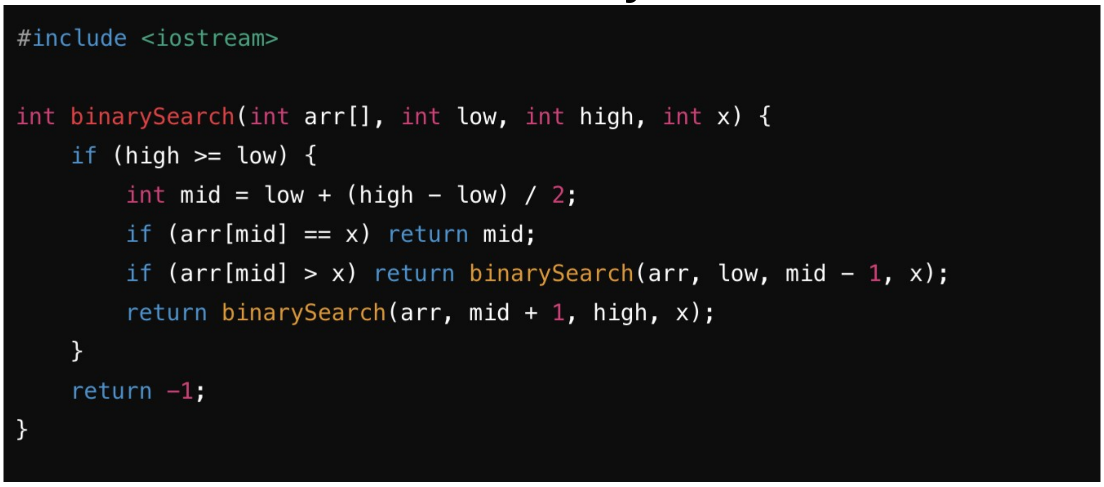
```cpp
#include <iostream>

#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5, 8, 9, 11};
    int key = 8;

    auto it = std::find(v.begin(), v.end(), key);

    if (it != v.end())
        std::cout << key << " Found at Position: " << it - v.begin() + 1;
    else
        std::cout << key << " NOT found.";

    return 0;
}
```

Iterative Binary Search



```cpp
#include <iostream>

int binarySearch(int arr[], int low, int high, int x) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] < x) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Recursive Binary Search



```cpp
#include <iostream>

int binarySearch(int arr[], int low, int high, int x) {
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return binarySearch(arr, low, mid - 1, x);
        return binarySearch(arr, mid + 1, high, x);
    }
    return -1;
}
```