Linked List

Question 1:

```
node *createLinkedList(int n)
{
// TO DO
   if(n==0)
      return NULL;

   node* head= new node;
   cin>>head->data;
   head->next = NULL;
   node* final_node= head;

   for(int i=0; i<n-1; i++){
   node* temp= new node;
   cin >> temp->data;
   temp->next=NULL;
   final_node->next = temp;
   final_node = temp;

   }

   return head;

}
bool isEqual(node *head1, node *head2)
{
 // TO DO
```

```
   while(head1 != NULL && head2 != NULL){
      if(head1->data != head2->data)
         return false;
      head1=head1->next;
      head2=head2->next;
   }

   if((head1 != NULL && head2 == NULL) || (head1 == NULL && head2 != NULL))
      return false;

   return true;

}
```

Question 2:

```
int countNode(node* head)
{
  //TODO
   int n=0;
   while(head != NULL){
      head= head->next;
      n++;
   }
   return n;
}
```

Question 3:

Question 4:

```
node *createLinkedList(int n)
{
```

```
 // TO DO
 if(n==0)
      return NULL;

   node* head= new node;
   cin>>head->data;
   head->next = NULL;



   for(int i=0; i<n-1; i++){
   node* temp= new node;
   cin >> temp->data;
   temp->next=head;
   head = temp;
   }

   return head;
}
```

Question 5:

```
node *createLinkedList(int n)
{
 // TO DO
   if(n==0)
      return NULL;

   node* head= new node;
   cin>>head->data;
   head->next = NULL;
```

```
    node* final_node= head;

    for(int i=0; i<n-1; i++){
    node* temp= new node;
    cin >> temp->data;
    temp->next=NULL;
    final_node->next = temp;
    final_node = temp;

    }

    return head;
}

node* addend(node* head, int n){
    node* temp= new node;
    temp->next = NULL;
    temp->data = n;

    if(head == NULL)
        head = temp;
    else{
        node* tmp=head;
        while(tmp->next != NULL)
            tmp=tmp->next;

        tmp->next = temp;
    }
```

```
    return head;
}

node* evenThenOddLinkedList(node *head)
{
  //TODO
  node* even_head = NULL;
    node* odd_head = NULL;

    while(head != NULL){
        int data = head->data;
        node* temp = head->next;
        delete head;
        head = temp;

        if(data %2 == 0)
            even_head = addend(even_head, data);
        else
            odd_head = addend(odd_head, data);
    }

    if(even_head == NULL)
        return odd_head;

    node* tmp= even_head;
    while(tmp->next != NULL)
        tmp= tmp->next;

    tmp->next = odd_head;
```

```
    return even_head;
}
```

Question 6:

```
node *insertNode(node* head, node* new_node, int pos)
{
  // TO DO
  if(pos<=0)
        return head;
    node* temp=head;

    if(head == NULL)
        head= new_node;

    if(pos == 1){
        temp->next= head;
        head = temp;
        return head;
    }

    pos--;
    while(temp->next != NULL && --pos)
        temp = temp->next;

    new_node->next = temp->next;
    temp->next = new_node;
```

```
    return head;
}
```

Question 7:

```
void mergeList(node* head1, node* head2)
{
  // TODO
  node* tmp= head1;
    while(tmp->next != NULL)
        tmp=tmp->next;

    tmp->next = head2;
}
```

Linked List + OOP: Revision

Question 1:

```
void BookList::addNewBook(Book* _pBook) {
    if (pHead == NULL) {
        pHead = _pBook;
        pTail = _pBook;
        return;
    }

    pTail->setNext(_pBook);
    pTail = pTail->getNext();
    return;
}
```

Question 2:

```
int BookList::findBook(char* name) {
```

```
    if (pHead == NULL)
        return -1;

    Book* temp = pHead;
    while (temp != NULL) {
        if (strcmp(name, temp->getName()) == 0 && bool(*temp) == false)
            return temp->getID();
        temp = temp->getNext();
    }

    return -1;
}
```

Question 3:

```
Book::Book(const char* _name, Author* au, bool isBorrowed, BookType type) {
    ID = Library::getNewID(type);
    name = (char*)_name;
    isBorrowed = false;
    pAuthor = au;
    pNext = NULL;
    typ = type;
}
```

Question 4:

```
Book::Book(Book& book) {
    ID = Library::getNewID(book.getType());
    name = book.getName();
    isBorrowed = *(&book);
    pAuthor = new Author(book.getAuthorName());
    pNext = book.getNext();
    typ = book.getType();
```

```
}
```

Question 5:

```
void Book::disp() {
   const char *name;
   name  = this->getName();
   const char *au;
   au = this->getAuthorName();
   cout << "ID: " << this->getID();
   cout << "\nBook: " << *name << " of " << *au << " is available right now.\n" ;
}
```

Question 6:

```
Book::operator bool() {
   return this->isBorrowed;
}
```

Linked List + OOP: Application

Question 1:

```
void Polynomial::insertTerm(const Term& term) {
   // STUDENT ANSWER
   if (term.coeff == 0)
      return;

   if (terms->size() == 0){
      terms->add(0, term);
      return;
   }

   int ind = 0;
   while ( terms->get(ind).exp > term.exp) {
```

```
        ind++;
        if (ind == terms->size()) {
            break;
        }
    }

    if((terms->size() == ind) || (terms->get(ind).exp < term.exp)) {
        terms->add(ind, term);
    } else {
        int coeff = term.coeff + terms->get(ind).coeff;
        terms->removeAt(ind);
        if (coeff != 0)
            insertTerm(coeff, term.exp);
    }
}

void Polynomial::insertTerm(double coeff, int exp) {
    // STUDENT ANSWER
    Term* term = new Term(coeff, exp);
    insertTerm(*term);
}
```

Question 2:

```
template <class T>
SLinkedList<T>::Iterator::Iterator(SLinkedList<T>* pList, bool begin)
{
    /*
        Constructor of iterator
        * Set pList to pList
        * begin = true:
```

```
   * * Set current (index = 0) to pList's head if pList is not NULL
   * * Otherwise set to NULL (index = -1)
   * begin = false:
   * * Always set current to NULL
   * * Set index to pList's size if pList is not NULL, otherwise 0
*/
this->pList = pList;
if (begin == true)
{
   if (this->pList != NULL)
   {
      index = 0;
      this->current = pList->head;
   }
   else
   {
      index = -1;
      this->current = NULL;
   }
}
else
{
   if (this->pList != NULL)
   {
      index = this->pList->size();
      this->current = NULL;
   }
   else
   {
```

```
            index = 0;
            this->current = NULL;
        }
    }
}

template <class T>
typename SLinkedList<T>::Iterator& SLinkedList<T>::Iterator::operator=(const Iterator& iterator)
{
    /*
        Assignment operator
        * Set this current, index, pList to iterator corresponding elements.
    */
    this->current = iterator.current;
    this->index = iterator.index;
    this->pList = iterator.pList;
    return *this;
}

template <class T>
void SLinkedList<T>::Iterator::remove()
{
    /*
        Remove a node which is pointed by current
        * After remove current points to the previous node of this position (or node with index - 1)
        * If remove at front, current points to previous "node" of head (current = NULL, index = -1)
        * Exception: throw std::out_of_range("Segmentation fault!") if remove when current is NULL
    */
    if (current == NULL)
```

```
    throw std::out_of_range("Segmentation fault!");

if (this->pList->head == current)
{
    this->current = NULL;
    this->index = -1;
    return;
}

Node *curr = this->pList->head->next;
Node *prev = this->pList->head;
while (curr != NULL)
{
    if (curr == this->current)
    {
        //Connect the next Node of Curr;
        prev->next = curr->next;
        //if curr is the last Node => re assign tail
        if (pList->tail == curr)
            pList->tail = prev;
        curr->next = NULL;
        this->index = -1;
        delete curr;
        this->current = prev;
        this->pList->count--;
        return;
    }
    curr = curr->next;
    prev = prev->next;
```

```
    }
}

template <class T>
void SLinkedList<T>::Iterator::set(const T& e)
{
    /*
        Set the new value for current node
        * Exception: throw std::out_of_range("Segmentation fault!") if current is NULL
    */
    if (current == NULL)
        throw std::out_of_range("Segmentation fault!");
    return current->data;
}

template <class T>
T& SLinkedList<T>::Iterator::operator*()
{
    /*
        Get data stored in current node
        * Exception: throw std::out_of_range("Segmentation fault!") if current is NULL
    */
    if (current == NULL)
        throw std::out_of_range("Segmentation fault!");
    return current->data;
}

template <class T>
bool SLinkedList<T>::Iterator::operator!=(const Iterator& iterator)
```

```
{
   /*
      Operator not equals
      * Returns true if two iterators points the same node and index
   */
   if ((this->current != iterator.current) && (this->index != iterator.index))
      return true;
   else
      return false;
}
// Prefix ++ overload
template <class T>
typename SLinkedList<T>::Iterator& SLinkedList<T>::Iterator::operator++()
{
   /*
      Prefix ++ overload
      * Set current to the next node
      * If iterator corresponds to the previous "node" of head, set it to head
      * Exception: throw std::out_of_range("Segmentation fault!") if iterator corresponds to the end
   */
   if (this->current->next == this->pList->head)
      this->current = this->pList->head;

   if (this->current == NULL)
      throw std::out_of_range("Segmentation fault!");

   current = current->next;
   return *this;
}
```

```
// Postfix ++ overload
template <class T>
typename SLinkedList<T>::Iterator SLinkedList<T>::Iterator::operator++(int)
{
   /*
      Postfix ++ overload
      * Set current to the next node
      * If iterator corresponds to the previous "node" of head, set it to head
      * Exception: throw std::out_of_range("Segmentation fault!") if iterator corresponds to the end
   */
   if (this->current->next == this->pList->head)
      this->current = this->pList->head;

   if (this->current == NULL)
      throw std::out_of_range("Segmentation fault!");

   Iterator iterator = *this;
   ++*this;
   return iterator;
}
```

Question 3:

```
template <class T>
void SLinkedList<T>::add(const T& e) {
   /* Insert an element into the end of the list. */
   Node *newNode = new Node(e, NULL);
   if (head == NULL)
   {
      head = newNode;
      tail = newNode;
```

```
    }
    else
    {
        tail->next = new Node(e, NULL);
        tail = tail->next;
    }
    count++;
}

template<class T>
void SLinkedList<T>::add(int index, const T& e) {
    /* Insert an element into the list at given index. */
    if (index == count || count == 0)
    {
        add(e);
    }
    else if (index == 0)
    {
        if (head == NULL)
        {
            head = new Node(e, NULL);
            tail = head;
        }
        else
        {
            Node *newNode = new Node(e, NULL);
            newNode->next = head;
            head = newNode;
        }
```

```
        count++;
    }
    else
    {
        Node *newNode = new Node(e, NULL);
        Node *ptr = head;
        for (int i = 0; i < index - 1; i++)
        {
            ptr = ptr->next;
        }
        if (ptr->next == NULL)
        {
            ptr->next = newNode;
            tail = newNode;
        }
        else
        {
            newNode->next = ptr->next;
            ptr->next = newNode;
        }
        count++;
    }
}

template<class T>
int SLinkedList<T>::size() {
    /* Return the length (size) of list */
    return count;
}
```

Question 4:

```
template<class T>
T SLinkedList<T>::get(int index) {
   /* Give the data of the element at given index in the list. */
   if (empty() == true || index < 0 || index > count - 1)
   {
      throw out_of_range("The index is out of range!");
   }
   Node *temp = head;
   for (int i = 0; i < index; i++)
   {
      temp = temp->next;
   }
   return (temp->data);
}


template <class T>
void SLinkedList<T>::set(int index, const T& e) {
   /* Assign new value for element at given index in the list */
   if (empty() == true || index > count)
      return;
   Node *temp = head;
   for (int i = 0; i < index; i++)
   {
      temp = temp->next;
   }
   temp->data = e;
}
```

```
template<class T>
bool SLinkedList<T>::empty() {
   /* Check if the list is empty or not. */
   return (count == 0);
}

template<class T>
int SLinkedList<T>::indexOf(const T& item) {
   /* Return the first index wheter item appears in list, otherwise return -1 */
   if (empty() == true)
      return -1;
   if (head->data == item)
      return 0;
   Node *temp = head;
   int pos = 0;
   while (temp != NULL)
   {
      if (temp->data == item)
      {
         return pos;
      }
      pos++;
      temp = temp->next;
   }
   return -1;
}

template<class T>
bool SLinkedList<T>::contains(const T& item) {
```

```
    /* Check if item appears in the list */
    if (empty() == true)
        return false;
    Node *ptr = head;
    if (ptr->data == item)
        return true;
    while (ptr != nullptr)
    {
        if (ptr->data == item)
            return true;
        ptr = ptr->next;
    }
    return false;
}
```

Question 5:

```
template <class T>
T SLinkedList<T>::removeAt(int index)
{
    /* Remove element at index and return removed value */
    T answer;
    if (empty() == true || index < 0 || index > count)
        throw out_of_range("The index is out of range!");
    if (index == 0)
    {
        Node *temp = head;
        answer = temp->data;
        head = head->next;
        delete temp;
    }
```

```
    else if (index == count - 1)
    {
        Node *ptr = head;
        while (ptr->next != tail)
            ptr = ptr->next;
        Node *temp = tail;
        answer = temp->data;
        tail = ptr;
        delete temp;
    }
    else
    {
        Node *prev = head;
        Node *curr = head->next;
        for (int i = 0; i < index - 1; i++)
        {
            prev = prev->next;
            curr = curr->next;
        }
        prev->next = curr->next;
        answer = curr->data;
        delete curr;
    }
    count--;
    return answer;
}

template <class T>
bool SLinkedList<T>::removeItem(const T& item)
```

```
{
   /* Remove the first apperance of item in list and return true, otherwise return false */
   if (empty() == true)
      return false;
   if (head->data == item){
     Node* del = head;
     head = head->next;
     delete del;
     count -=1;
     return true;
   }
   else if (tail->data == item){
      Node *prev = head;
      Node *curr = head->next;
      while (curr->next != NULL)
      {

         prev = prev->next;
         curr = curr->next;
      }
      delete curr;
      tail = prev;
      count-=1;
      return true;
   }
   else {
      Node* tmp = head;
      while (tmp->next){
        if (tmp->next->data == item){
```

```
            Node* del = tmp->next;
            tmp->next = tmp->next->next;
            delete del;
            count -=1;
            return true;
          }
          tmp = tmp->next;
        }
    }
    return false;
}

template<class T>
void SLinkedList<T>::clear(){
    /* Remove all elements in list */
    this->count = 0;
    if (empty() == true)
        return;
    else
    {

        while (head->next != tail)
        {
            Node *temp = head->next;
            head = head->next;
            delete temp;
            count--;
        }
        tail->next = head;
```

```
    }
}
```

Stack and Queue

Question 1:

```
bool bfs(vector<vector<int>>& graph, vector<int>& color, int src) {
    queue<int> q;
    q.push(src);
    color[src] = 0;
    while (q.empty() == false) {
        int x = q.front();
        q.pop();

        int ad_ver = graph[x].size();
        for (int i =0; i < ad_ver; i++) {
            int y = graph[x][i];
            if (color[x] == 0 && color[y] == 0) {
                return false;
            } else if (color[x] == 1 && color[y] == 1) {
                return false;
            } else if (color[x] == 0 && color[y] == -1) {
                q.push(y);
                color[y] = 1;
            } else if (color[x] == 1 && color[y] == -1) {
                q.push(y);
                color[y] = 0;
            }

        }
```

```
    }
    return true;
}

bool isBipartite(vector<vector<int>>& graph) {
    int V = graph.size();
    vector<int> color(V, -1);
    int src = 0;
    for(; src < V; src++){
        if (graph[src].size() != 0 && color[src] == -1 && bfs(graph, color, src) == false)
            return false;
    }
    return true;
}
```

Question 2:

```
void bfs(vector<vector<int>> graph, int start) {
            int V = graph.size();
    vector<int> tra(V, 0);
    queue<int> q;
    q.push(start);
    tra[start] = 1;
    while (q.empty() == false) {
        int x = q.front();
        q.pop();
        cout  <<  x  <<  " ";
        int size = graph[x].size();
        for (int i = 0; i < size; i++) {
            int y = graph[x][i];
```

```
            if (tra[y] == 0) {
                q.push(y);
                tra[y] = 1;
            }
        }
    }
}
```

Question 3: QUEUE

```
void push(T item) {
    // TODO: Push new element into the end of the queue
    list.add(item);
}

T pop() {
    // TODO: Remove an element in the head of the queue
    T backup = list.get(0);
    list.removeAt(0);
    return backup;
}

T top() {
    // TODO: Get value of the element in the head of the queue
    return list.get(0);
}

bool empty() {
    // TODO: Determine if the queue is empty
    return list.empty();
}
```

```
int size() {
   // TODO: Get the size of the queue
   return list.size();
}

void clear() {
   // TODO: Clear all elements of the queue
   return list.clear();
}
```

Question 4: STACK

```
void push(T item) {
   // TODO: Push new element into the top of the stack
   return list.add(0, item);
}

T pop() {
   // TODO: Remove an element on top of the stack
   return list.removeAt(0);
}

T top() {
   // TODO: Get value of the element on top of the stack
   return list.get(0);
}

bool empty() {
   // TODO: Determine if the stack is empty
   return list.empty();
```

```
}

int size() {
   // TODO: Get the size of the stack
   return list.size();
}

void clear() {
   // TODO: Clear all elements of the stack
   return list.clear();
}
```

Question 5:

```
string removeDuplicates(string S){
   /*TODO*/
   stack<char> s;
   int L = S.size();
   for(int i = 0; i < L; i++) {
      if (s.empty() == false && s.top() == S[i])
         s.pop();
      else
         s.push(S[i]);
   }
   string res = "";
   while(s.empty() == false) {
      char x = s.top();
      s.pop();

      res.insert(res.begin(), x);
   }
```

```
    return res;
}
```

Question 6:

```
bool corres(char a, char b){
        if (a == '(' && b == ')')
            return true;
        if (a == '{' && b == '}')
            return true;
        if (a == '[' && b == ']')
            return true;
        return false;
    }
    bool isValidParentheses(string S) {
        stack<char> s;
        int L = S.size();
        for(int i = 0; i < L; i++) {
            if (s.empty() == false && corres(s.top(), S[i]))
                s.pop();
            else
                s.push(S[i]);
        }

        return s.empty();
    }
```

SORTING ALGORITHM

Question 1:

```
template <class T>
```

```
void SLinkedList<T>::bubbleSort()
{

   for ( int  i = 0; i < this->size()-1; i++) {
      Node* temp = head;
      for ( int j = this->size()-i-1; j > 0; j--) {
         if ( temp->data > temp->next->data) {
            T x = temp->data;
            temp->data = temp->next->data;
            temp->next->data = x;
         }
         temp = temp->next;
      }
      this->printList();
   }
}
```

Question 2:

```
bool isPermutation (string a, string b) {
   //TODO
   if (a.size() != b.size())
      return false;
   int size = a.size();
   for (int i = 0; i < size-1; i++)
      for (int j = 0; j < size-i-1; j++)
         if (a[j] > a[j+1])
            swap(a[j], a[j+1]);
   //cout<<a;
   for (int i = 0; i < size-1; i++)
      for (int j = 0; j < size-i-1; j++)
```

```
            if (b[j] > b[j+1])
                swap(b[j], b[j+1]);
    //cout<<b;
    return a==b;

}
```

Question 3:

```
T *Sorting<T>::Partition(T *start, T *end)
{
}

template <class T>
void Sorting<T>::insertionSort(T *start, T *end)
{
    cout  <<  "Insertion sort: ";
    printArray(start, end);
    int size = start-end+1;
    if (size <= 1)
        return;
    for (int i = 1; i < size; i++) {
        int key = start[i];
        int j = i;
        while (start[j] > key) {
            start[j] = start[j-1];
            j--;
        }
        start[j-1] = key;
    }
}
```

```
template <class T>
void Sorting<T>::hybridQuickSort(T *start, T *end, int min_size)
{
   int size = end - start;
   //cout<<"\nSize: "<<size<<endl;
   if (size <= 0)
      return;

   if (size < min_size)
     return Sorting<T>::insertionSort(start, end);
   cout  <<  "Quick sort: ";
   printArray(start, end);
   int key = start[0];
   int i = 1;
   int j = size-1;
   while ( j > i) {
      while (start[i] < key)
         i++;
      while (start[j] > key)
         j--;
      if( j > i)
         swap(start[i], start[j]);
   }

   swap(start[j], start[0]);

   hybridQuickSort(start, start+j, min_size);
   hybridQuickSort(start+j+1, end, min_size);
```

```
}
```

Question 4:

```
static void merge(T* left, T* middle, T* right){
   /*TODO*/
   if (left == right)
      return;
   int S1 = middle - left;
   int S2 = right - middle +1;
   int LA[S1];
   int RA[S2];
   for(int i=0; i<S1; i++)
      LA[i] = left[i];
   // cout<<S1<<": ";
   // for(int i=0; i<S1; i++)
   //     cout<<LA[i]<<" ";
   // cout<<endl<<endl;
   for(int j=0; j<S2; j++)
      RA[j] = middle[j];
   // cout<<S2<<": ";
   // for(int j=0; j<S2; j++)
   //     cout<<RA[j]<<" ";
   // cout<<endl<<endl;
   int i = 0;
   int j = 0;
   int k = 0;
   while (i<S1 && j<S2) {
      if (LA[i] < RA[j]) {
         left[k] = LA[i];
         i++;
```

```
            k++;
        } else {
            left[k] = RA[j];
            j++;
            k++;
        }
    }

    if (i == S1) {
        while (j < S2) {
            left[k] = RA[j];
            k++;
            j++;
        }
    } else if (j == S2) {
        while (i < S1) {
            left[k] = LA[i];
            i++;
            k++;
        }
    }
    Sorting::printArray(left, right);
}
static void mergeSort(T* start, T* end){
    /*TODO*/
    int S = end - start;
    if (S <= 0)
        return;
```

```
    mergeSort(start, start+S/2);
    mergeSort(start+S/2+1, end);
    merge(start, start+S/2+1, end);
}
```

Question 5:

```
template <class T>
void Sorting<T>::oddEvenSort(T *start, T *end)
{
    /*TODO*/
    int n = end - start;
    bool isSorted = false;
    while (isSorted == false) {
        isSorted = true;

        for(int i = 0; i <= n-2 ; i+=2) {
            if (start[i] > start[i+1]) {
                swap(start[i], start[i+1]);
                isSorted = false;
            }
        }

        for(int i = 1; i <= n-2; i+=2) {
            if (start[i] > start[i+1]) {
                swap(start[i], start[i+1]);
                isSorted = false;
            }
        }
        printArray(start, end);
    }
```

```
}
```

Question 6:

```
static T* Partition(T* start, T* end) {
   // TODO: return the pointer which points to the pivot after rearrange the array.
   int S = end-start;
   if (S<=0)
      return 0;
   int i = 0, j = S-1;
   T pivot = start[0];
   while (i<j) {
      while (start[i] <= pivot)
         i++;
      while (start[j] > pivot)
         j--;
      swap(start[i], start[j]);
   }

   swap(start[i], start[j]);
   swap(start[0], start[j]);
   cout<<j<<" ";
   return &start[j];
}

static void QuickSort(T* start, T* end) {
   // TODO
   // In this question, you must print out the index of pivot in subarray after everytime calling method Partition.
   T* pivot = Partition(start, end);
   if (pivot) {
```

```
        QuickSort(start, pivot);
        QuickSort(pivot+1, end);
    }
}
```

Question 7:

```
static void sortSegment(T* start, T* end, int segment_idx, int cur_segment_total) {
    // TODO
    int S = end - start;
    for (int i = segment_idx; i < S-cur_segment_total; i+=cur_segment_total) {
        int j = i+cur_segment_total;
        int key = start[j];
        while (key < start[j-cur_segment_total]) {
            start[j] = start[j-cur_segment_total];
            j -= cur_segment_total;
            if (j < cur_segment_total)
                break;
        }
        start[j] = key;
    }
}


static void ShellSort(T* start, T* end, int* num_segment_list, int num_phases) {
    // TODO
    // Note: You must print out the array after sorting segments to check whether your algorithm is true.
    for (int i = num_phases-1; i >= 0; i--) {
        for (int j = 0; j < num_segment_list[i]; j++) {
            sortSegment(start, end, j, num_segment_list[i]);
        }
        cout<<num_segment_list[i]<<" segments: ";printArray(start, end);
```

```
    }
}
```

Question 8:

```
template <class T>
void Sorting<T>::selectionSort(T *start, T *end)
{
int S = end-start;
    for(int i = 0; i < S-1; i++) {
        int min_ind = i;
        for (int j = i; j < S; j++) {
            if (start[j] < start[min_ind])
                min_ind = j;
        }
        swap(start[i], start[min_ind]);
        printArray(start, end);
    }
}
```

Question 9:

```
static void merge(T* left, T* middle, T* right){
    /*TODO*/
    if (left == right)
        return;
    int S1 = middle - left;
    int S2 = right - middle;
    int LA[S1];
    int RA[S2];
    for(int i=0; i<S1; i++)
        LA[i] = left[i];
```

```
for(int j=0; j<S2; j++)
    RA[j] = middle[j];

int i = 0;
int j = 0;
int k = 0;
while (i<S1 && j<S2) {
    if (LA[i] < RA[j]) {
        left[k] = LA[i];
        i++;
        k++;
    } else {
        left[k] = RA[j];
        j++;
        k++;
    }
}

if (i == S1) {
    while (j < S2) {
        left[k] = RA[j];
        k++;
        j++;
    }
} else if (j == S2) {
    while (i < S1) {
        left[k] = LA[i];
        i++;
        k++;
```

```
        }
    }
}

static void insertionSort(T* start, T* end) {
    // TODO
    int S = end-start;
    if (S <= 1)
        return;
    for (int i = 1; i < S; i++) {
        int key = start[i];
        int j = i;
        while (start[j-1] > key) {
            start[j] = start[j-1];
            j--;
            if (j == 0)
                break;
        }
        start[j] = key;
    }
}

static void TimSort(T* start, T* end, int min_size) {
    // TODO
    // You must print out the array after using insertion sort and everytime calling method merge.
    int S = end-start;
    for (T* begin = start; begin < end; begin+=min_size) {
        insertionSort(begin, min(end, begin+min_size));
    }
```

```
    cout << "Insertion Sort: ";
    printArray(start, end);

    int i = 1;
    for (int jump = min_size; jump < S; jump*=2) {
        for (T* base = start; base < end; base+=2*jump) {
            merge(base, min(base+jump, end), min(base+2*jump, end));
            cout << "Merge " << i << ": "; i++;
            printArray(start, end);
        }
    }
}
```

BST

Question 1:

```
Node* add(Node* pRoot, T value) {
        if (pRoot == NULL) return new Node(value);
        if (value < pRoot->value)
            pRoot->pLeft = this->add(pRoot->pLeft, value);
        else
            pRoot->pRight = this->add(pRoot->pRight, value);
        return pRoot;
    }

    Node* minValue(Node* node) {
        Node* current = node;
        while(current && current->pLeft != NULL)
            current = current->pLeft;
```

```
    return current;
}

Node* remove(Node* root, T value) {
    if (root == NULL)
        return root;

    if (value < root->value)
        root->pLeft = remove(root->pLeft, value);
    else if (value > root->value)
        root->pRight = remove(root->pRight, value);
    else {
        if (root->pLeft == NULL and root->pRight == NULL)
            return NULL;

        else if (root->pLeft == NULL) {
            Node* temp = root->pRight;
            free(root);
            return temp;
        }
        else if (root->pRight == NULL) {
            Node* temp = root->pLeft;
            free(root);
            return temp;
        }

        Node* temp = minValue(root->pRight);

        root->value = temp->value;
```

```
            root->pRight = remove(root->pRight, temp->value);
        }
        return root;
    }


    void add(T value){
        //TODO
        this->root = add(this->root, value);
    }

    void deleteNode(T value){
        //TODO
        this->root = remove(this->root, value);
    }
```

Question 2:

```
int diameterOpt(Node* root, int* h) {
        int lh = 0, rh = 0;
        int ld = 0, rd = 0;

        if (root == NULL) {
            *h = 0;
            return 0;
        }

        ld = diameterOpt(root->pLeft, &lh);
        rd = diameterOpt(root->pRight, &rh);

        *h = max(lh, rh) + 1;
```

```
    return max(lh+rh+1, max(ld, rd));
  }

  int getDiameter() {
    int height = 0;
    return diameterOpt(this->root, &height);
  }
```

Question 3:

```
void printCurrentLevel(Node* root, int level) {
    if (root == NULL)
        return;
    if (level == 1)
        cout << root->value << " ";
    else if (level > 1) {
        printCurrentLevel(root->pLeft, level-1);
        printCurrentLevel(root->pRight, level-1);
    }
  }

  int height(Node* node) {
    if (node == NULL) return 0;
    else {
        int lheight = height(node->pLeft);
        int rheight = height(node->pRight);
        if(lheight > rheight) return (lheight+1);
        else return (rheight+1);
    }
  }
```

```
  void BFS() {
     int h = height(this->root);
     int i;
     for (i = 1; i <= h; i++)
        printCurrentLevel(this->root, i);
  }
```

Question 4:

```
int TwoChildrenNode(Node* root) {
     if (root == NULL) {
        return 0;
     }
     if (root->pLeft != NULL && root->pRight != NULL) {
        return 1+TwoChildrenNode(root->pLeft) +TwoChildrenNode(root->pRight);
     }
     return TwoChildrenNode(root->pLeft) + TwoChildrenNode(root->pRight);
  }

  int countTwoChildrenNode() {
     return TwoChildrenNode(this->root);
  }
```

Question 5:

```
bool isBSThelper(Node* root) {
     if (root == NULL) {
        return true;
     }

     if (root->pLeft != NULL && root->pLeft->value > root->value)
        return false;
```

```
    if (root->pRight != NULL && root->pRight->value < root->value)
        return false;

    if (!isBSThelper(root->pLeft) || !isBSThelper(root->pRight))
        return false;

    return 1;
  }

  bool isBST() {
    return isBSThelper(this->root);
  }
```

Question 6:

```
bool contains(Node* root, T value) {
    if (root == NULL)
        return false;
    else if (root->value == value)
        return true;
    else
        return contains(root->pLeft, value) || contains(root->pRight, value);
  }

  void sumhelper(T& ans, Node* node, T l, T r) {
    if (node != NULL) {
        if (l <= node->value && node->value <= r)
            ans+=node->value;
        sumhelper(ans, node->pLeft, l, r);
        sumhelper(ans, node->pRight, l, r);
```

```
    }
  }

  void helper(Node* root, int& ans) {
    if (!root)
      return;
    if (!root->pLeft && !root->pRight)
      ans += root->value;
    helper(root->pLeft, ans);
    helper(root->pRight, ans);
  }

  int sumOfLeafs() {
    int ans = 0;
    helper(this->root, ans);
    return ans;
  }

  bool find(T i) {
    return contains(this->root, i);
  }

  T sum(T l, T r) {
    T ans = 0;
    sumhelper(ans, this->root, l, r);
    return ans;
  }
```

Question 7:

```
T getMin() {
```

```
        Node* cur = this->root;
        while (cur->pLeft != NULL) {
            cur = cur->pLeft;
        }

        return cur->value;
    }

    T getMax() {
        Node* cur = this->root;
        while (cur->pRight != NULL) {
            cur = cur->pRight;
        }

        return cur->value;
    }
```

Question 8:

```
int height(Node* node) {
        if (node == NULL) return 0;
        else {
            int lheight = height(node->pLeft);
            int rheight = height(node->pRight);
            if(lheight > rheight) return (lheight+1);
            else return (rheight+1);
        }
    }
void printPreorder(Node* node) {
        if (node == NULL)
            return;
```

```
    cout << node->value << " ";
    printPreorder(node->pLeft);
    printPreorder(node->pRight);
}

void printInorder(Node* node) {
    if (node == NULL)
        return;
    printInorder(node->pLeft);
    cout << node->value << " ";
    printInorder(node->pRight);
}

int getHeight() {
    return height(this->root);
}
void printPostorder(Node* node) {
    if (node == NULL)
        return;

    printPostorder(node->pLeft);
    printPostorder(node->pRight);
    cout << node->value << " ";
}

string preOrder() {
    string s;
    printPreorder(this->root);
```

```
        return s;
    }

    string inOrder() {
        string s;
        printInorder(this->root);
        return s;
    }

    string postOrder() {
        string s;
        printPostorder(this->root);
        return s;
    }
```

Question 9:

```
void helper(Node* root, int& ans) {
        if (!root)
            return;
        if (!root->pLeft && !root->pRight)
            ans += root->value;
        helper(root->pLeft, ans);
        helper(root->pRight, ans);
    }

    int sumOfLeafs() {
        int ans = 0;
        helper(this->root, ans);
        return ans;
    }
```

Heap

Question 1:

```
static void sift_down(T* start, int i, int n) {
   T tmp{ *(start + i) };
   T child{};
   while (2*i + 1 < n) {
      child = 2 * i + 1;
      if (child != n - 1 && *(start + child) < *(start + child + 1)) {
         ++child;
      }
      if (tmp < *(start + child)) {
         *(start + i) = *(start + child);
         i = child;
      }
      else break;
   }
   *(start + i) = tmp;
}
static void heapSort(T* start, T* end){
   //TODO
   int size{ static_cast<int>(end - start) };
   for (int i{ size / 2 - 1 }; i >= 0; --i) {
      sift_down(start, i, size);
   }
   for (int j{ size - 1 }; j > 0; --j) {
      T tmp{ *(start + j) };
      *(start + j) = *start;
      *start = tmp;
```

```
        sift_down(start, 0, j);
    }
    Sorting<T>::printArray(start,end);
}
```

Question 2:

```
void swap(int* a, int* b) {
    if (a != b) {
        int tmp{ *a };
        *a = *b;
        *b = tmp;
    }
}

int minWaitingTime(int n, int arrvalTime[], int completeTime[]) {
    int time{ 0 };
    int totalWaitingTime{ 0 };
    int count{ n };
    while (count > 0) {
        int target{ -1 };
        int minTime{ 99999 };
        for (int i{ 0 }; i < count; ++i) {
            if (arrvalTime[i] <= time && completeTime[i] < minTime) {
                target = i;
                minTime = completeTime[i];
            }
        }
        if (target >= 0) {
            totalWaitingTime += completeTime[target] + time - arrvalTime[target];
            time += completeTime[target];
```

```
            swap(&arrvalTime[target], &arrvalTime[count - 1]);
            swap(&completeTime[target], &completeTime[count - 1]);
            --count;
        }
        else ++time;
    }
    return totalWaitingTime;
}
```

Question 3:

```
template<class T>
int Heap<T>::size(){
    return this->count;
}

template<class T>
bool Heap<T>::isEmpty(){
    return this->count == 0;
}

template<class T>
T Heap<T>::peek(){
    if (count == 0) return -1;
    return elements[0];
}

template<class T>
bool Heap<T>::contains(T item){
    for (int i = 0; i < count; i++)
        if (elements[i] == item)
```

```
            return true;

    return false;
}

template<class T>
bool Heap<T>::pop(){
    if (count == 0)
    return false;
    swap(elements[0], elements[count-1]);
    count --;
    reheapDown(0);
    return true;
}
```

Question 4:

```
template<class T>
void Heap<T>::push(T item){
    if (count == capacity)
        ensureCapacity(3 * capacity);
    elements[count++] = item;
    reheapUp(count - 1);
}

template<class T>
void Heap<T>::ensureCapacity(int minCapacity){
    T* n_heap{ new T[minCapacity] };
    for (int i{ 0 }; i < count; ++i){
        n_heap[i] = elements[i];
    }
```

```
    delete[] elements;
    elements = n_heap;
    capacity = minCapacity;
}

template<class T>
void Heap<T>::reheapUp(int position){
    if (position < count && position > 0) {
        T tmp{ elements[position] };
        while ((position - 1) >= 0) {
            if (tmp > elements[(position - 1) / 2]) {
                elements[position] = elements[(position - 1) / 2];
                position = (position - 1) / 2;
            }
            else break;
        }
        if (elements[position] != tmp) elements[position] = tmp;
    }
}
```

Question 5:

```
void reheapDown(int maxHeap[], int numberOfElements, int index)
{
    int tmp{ maxHeap[index] };
    int child{};
    while (2 * index + 1 < numberOfElements) {
        child = 2 * index + 1;
        if (child != numberOfElements - 1 && maxHeap[child] < maxHeap[child + 1]) ++child;
        if (tmp < maxHeap[child])
            maxHeap[index] = maxHeap[child];
```

```
        else break;
        index = child;
    }
    if (maxHeap[index] != tmp) maxHeap[index] = tmp;
}

void reheapUp(int maxHeap[], int numberOfElements, int index)
{
    if (index > 0 && index < numberOfElements) {
        int tmp{ maxHeap[index] };
        while (index - 1 >= 0) {
            if (tmp > maxHeap[(index - 1) / 2]) {
                maxHeap[index] = maxHeap[(index - 1) / 2];
                index = (index - 1) / 2;
            }
            else break;
        }
        if (maxHeap[index] != tmp) maxHeap[index] = tmp;
    }
}
```

Question 6:

```
template<class T>
int Heap<T>::getItem(T item) {
    // TODO: return the index of item in heap
    for (int i{ 0 }; i < count; ++i) {
        if (elements[i] == item) return i;
    }
    return -1;
}
```

```
template<class T>
void Heap<T>::remove(T item) {
   // TODO: remove the element with value equal to item
   int i{ getItem(item) };
   if (i != -1) {
      elements[i] = elements[count - 1];
      --count;
      if (i - 1 >= 0 && elements[i] > elements[(i - 1) / 2]) reheapUp(i);
      else reheapDown(i);
   }
}

template<class T>
void Heap<T>::clear() {
   // TODO: delete all elements in heap
   count = 0;
}
```

AVL

Question 1:

```
int BL(Node* root) {
   if (root == NULL)
      return 0;
   return getHeightRec(root->pLeft) - getHeightRec(root->pRight);
}

Node* Rot_R(Node* root) {
   Node* old_root = root;
```

```
    Node* new_root = root->pLeft;
    old_root->pLeft = new_root->pRight;
    new_root->pRight = old_root;
    return new_root;
}

Node* Rot_L(Node* root) {
    Node* old_root = root;
    Node* new_root = root->pRight;
    old_root->pRight = new_root->pLeft;
    new_root->pLeft = old_root;
    return new_root;
}

Node* reBalance(Node* root) {
    int balance = BL(root);
    //cout<<balance<<endl;
    if (balance > 1) {
        int bll = BL(root->pLeft);
        if (bll > 0) {
            return Rot_R(root);
        } else {
            root->pLeft = Rot_L(root->pLeft);
            return Rot_R(root);
        }
    } else if (balance < -1) {
        int blr = BL(root->pRight);
        if (blr < 0) {
            return Rot_L(root);
```

```
        } else {
            root->pRight = Rot_R(root->pRight);
            return Rot_L(root);
        }
    }

    return root;
}

Node* remove(Node* root, T value) {
    if (root == NULL)
        return 0;
    if (root->data > value) {
        root->pLeft = remove(root->pLeft, value);
    } else if (root->data < value) {
        root->pRight = remove(root->pRight, value);
    }else if (root->data == value) {
        if (root->pLeft == NULL && root->pRight == NULL) {
            delete root;
            return NULL;
        } else {
            Node* temp = (root->pLeft) ? root->pLeft:root->pRight;
            while(temp->pRight != NULL)
                temp = temp->pRight;
            root->data = temp->data;
            if (root->pLeft)
                // Sai root = remove...
                // Sua lai: root->pLeft = remove...
                root->pLeft = remove(root->pLeft, temp->data);
```

```
        else
            root->pRight = remove(root->pRight, temp->data);
      }
   }

   return reBalance(root);
}

void remove(const T &value) {
   //TODO
   this->root = remove(this->root, value);
}
```

Question 2:

```
int BL(Node* root) {
   if (root == NULL)
      return 0;
   return getHeightRec(root->pLeft) - getHeightRec(root->pRight);
}

Node* Rot_R(Node* root) {
   Node* old_root = root;
   Node* new_root = root->pLeft;
   old_root->pLeft = new_root->pRight;
   new_root->pRight = old_root;
   return new_root;
}

Node* Rot_L(Node* root) {
   Node* old_root = root;
```

```
    Node* new_root = root->pRight;
    old_root->pRight = new_root->pLeft;
    new_root->pLeft = old_root;
    return new_root;
}

Node* reBalance(Node* root) {
    int balance = BL(root);
    //cout<<balance<<endl;
    if (balance > 1) {
        int bll = BL(root->pLeft);
        if (bll > 0) {
            return Rot_R(root);
        } else {
            root->pLeft = Rot_L(root->pLeft);
            return Rot_R(root);
        }
    } else if (balance < -1) {
        int blr = BL(root->pRight);
        if (blr < 0) {
            return Rot_L(root);
        } else {
            root->pRight = Rot_R(root->pRight);
            return Rot_L(root);
        }
    }

    return root;
}
```

```
Node* remove(Node* root, T value) {
   if (root == NULL)
      return 0;
   if (root->data > value) {
      root->pLeft = remove(root->pLeft, value);
   } else if (root->data < value) {
      root->pRight = remove(root->pRight, value);
   }else if (root->data == value) {
      if (root->pLeft == NULL && root->pRight == NULL) {
         delete root;
         return NULL;
      } else {
         Node* temp = (root->pLeft) ? root->pLeft:root->pRight;
         while(temp->pRight != NULL)
            temp = temp->pRight;
         root->data = temp->data;
         if (root->pLeft)
            // Sai root = remove...
            // Sua lai: root->pLeft = remove...
            root->pLeft = remove(root->pLeft, temp->data);
         else
            root->pRight = remove(root->pRight, temp->data);
      }
   }

   return reBalance(root);
}
```

```
Node* insert(Node* root, T value) {
   if (root == NULL) {
      root = new Node(value);
      return root;
   }

   if (value >= root->data) {
      root->pRight = insert(root->pRight, value);
   } else {
      root->pLeft = insert(root->pLeft, value);
   }

   return reBalance(root);
}

void insert(const T &value){
   //TODO
   this->root = insert(this->root, value);
}
```

Question 3:

```
void Inorder(Node* root) {
   if (root != NULL) {
      Inorder(root->pLeft);
      cout << root->data<<" ";
      Inorder(root->pRight);
   }
}
```

```
void printInorder() {
   this->Inorder(this->root);
}

bool search(Node* root, T value) {
   if (root == NULL)
      return false;

   if (root->data == value) {
      return true;
   }

   if (value < root->data) {
      return search(root->pLeft, value);
   } else {
      return search(root->pRight, value);
   }
}

bool search(const T &value) {
   return search(this->root, value);
}

void clear(){
}
```

Hash Search

Question 1:

```
int binarySearch(int arr[], int left, int right, int x)
{
   if (right < left)
      return -1;
   int mid = (right+left)/2;
   cout << "We traverse on index: " << mid << endl;
   if (arr[mid] == x)
      return mid;
   else if (arr[mid] < x)
      return binarySearch(arr, mid+1, right, x);

   return binarySearch(arr, left, mid-1, x);
}
```

Question 2:

```
bool compare(pair<int, int>& pair1, pair<int, int>& pair2) {
   return (pair1.first + pair1.second) < (pair2.first + pair2.second);
}

bool findPairs(int arr[], int n, pair<int,int>& pair1, pair<int, int>& pair2)
{
  // TODO: If there are two pairs satisfy the condition, assign their values to pair1, pair2 and return true. Otherwise, return false.
  vector<pair<int, int>> vec;
  for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++){
         vec.push_back(pair<int, int>(arr[i], arr[j]));
      }
  }
```

```
  sort(vec.begin(), vec.end(), compare);

  auto it = vec.begin()+1;
  while (it != vec.end()) {
      if ( ((*it).first + (*it).second) == ((*(it-1)).first + (*(it-1)).second)) {
          pair1 = (*(it-1));
          pair2 = (*it);
          return true;
      }
      it++;
  }

  return false;
}
```

Question 3:

```
int foldShift(long long key, int addressSize)
{
   string address = to_string(key);
   int ret = 0;
   for (int i = 0; i < (int)address.size()-addressSize+1; i+=addressSize) {
      while (stoi(address.substr(i, 2))==0) {
         i++;
         if (i > (int)address.size()-addressSize)
            break;
      }
      ret += stoi(address.substr(i, 2));

   }
   return ret;
```

```
}

int rotation(long long key, int addressSize)
{
   int retval = foldShift(key, addressSize);
   string ret = to_string(retval);
   reverse(ret.begin(), ret.end());
   return stoi(ret);
}
```

Question 4:

```
int interpolationSearch(int arr[], int left, int right, int x)
{
   int pos = 0;


   if (left <= right && x >= arr[left] && x <= arr[right]) {


      pos = left
           + (((double)(right - left) / (arr[right] - arr[left]))
             * (x - arr[left]));

      cout << "We traverse on index: " << pos << endl;


      if (arr[pos] == x)
         return pos;
```

```
    if (arr[pos] < x)
        return interpolationSearch(arr, pos + 1, right, x);



    if (arr[pos] > x)
        return interpolationSearch(arr, left, pos - 1, x);
  }
  return -1;
}
```

Question 5:

```
int jumpSearch(int arr[], int x, int n) {
  // TODO: print the traversed indexes and return the index of value x in array if x is found, otherwise, return -1.
  int step = sqrt(n);
  int i;
  for(i = 0; i < n; i+=step)
  {
    cout << i <<" ";
    if(arr[i]>=x)
      break;
  }
  if(arr[i] == x)
    return i;
  for(int j = i-step+1; j < i; j++)
  {
    cout << j <<" ";
    if(arr[j] == x)
      return j;
  }
```

```
    return -1;


}
```

Question 6:

```
long int midSquare(long int seed)
{
    string key = to_string(seed*seed);
    int size = (int)key.size();
    return stoi(key.substr(max(0, size-6), 4));


}
long int moduloDivision(long int seed, long int mod)
{
    return seed%mod;
}
long int digitExtraction(long int seed,int* extractDigits,int size)
{
    string num = to_string(seed);
    string ss;
    for (int i = 0; i < size; i++) {
        ss.push_back(num[extractDigits[i]]);
    }


    return stoi(ss);
}
```