- Stack is a linear data type that follows LIFO
- Main operations:
  - Push → add to top of stack → first elem to be removed
  - Pop → remove from the top of the stack
- Applications:
  - expression evaluation
  - backtracking algorithms
  - undo navigation
- underflow → throw error when pop from empty stack
- overflow → throw error when push to full stack

SLList:

```
template <typename T>
void SLList<T>::clear() {
   Node *current = head;
   while (current != nullptr) {
      Node *next = current->next;
      delete current;
      current = next;
   }
   head = tail = nullptr;
   size = 0;
}

template <typename T>
void SLList<T>::push_back(const T& value) {
   Node *p = new Node(value);
   if (empty()) {
      head = tail = p;
   } else {
      tail->next = p;
      tail = p;
   }
   size++;
}

template <typename T>
void SLList<T>::pop_front() {
   if (empty()) {
      throw std::out_of_range("List is empty");
   }
   Node *p = head;
   head = head->next;
   delete p;
   size--;
   if (empty()) {
      tail = nullptr;
   }
```

DLList:

```
DoublyLinkedList::DoublyLinkedList() {
  // TODO: Implement the constructor
  head = tail = nullptr;
  n_nodes = 0;
}

void DoublyLinkedList::push_back(int v) {
  // TODO: Implement the push_back method
  Node *newNode = new Node(v);
  if (empty()) {
    head = tail = newNode;
  } else {
    tail->next = newNode;
    newNode->prev = tail;
    tail = newNode;
  }
  n_nodes++;
}

void DoublyLinkedList::push_front(int v) {
  // TOOD: Implement the push_front method
  Node *newNode = new Node(v);
  if (empty()) {
    head = tail = newNode;
  } else {
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
  }
  n_nodes++;
}

void DoublyLinkedList::push_at(int idx, int v) {
  // TOOD: Implement the push_at method

  if (idx < 0 || idx > n_nodes) {
    throw std::out_of_range("Index out of range");
  }
  if (idx == 0) {
    push_front(v);
    return;
  }
  if (idx == n_nodes) {
    push_back(v);
    return;
  }
```

```
  Node *newNode = new Node(v);
  Node *current = head;
  for (int i = 0; i < idx; i++) {
    current = current->next;
  }
  newNode->prev = current->prev;
  newNode->next = current;
  current->prev->next = newNode;
  current->prev = newNode;
  n_nodes++;
}

void DoublyLinkedList::pop_back() {
  // TOOD: Implement the pop_back method
  if (empty()) {
    throw std::runtime_error("List is empty");
  }

  Node *toPop = tail;

  if (n_nodes == 1) {
    head = tail = nullptr;
  } else {
    tail = tail->prev;
    tail->next = nullptr;
  }

  delete toPop;
  n_nodes--;
}

void DoublyLinkedList::pop_front() {
  // TOOD: Implement the pop_front method
  if (empty()) {
    throw std::runtime_error("List is empty");
  }

  Node *toPop = head;

  if (n_nodes == 1) {
    head = tail = nullptr;
  } else {
    head = head->next;
    head->prev = nullptr;
  }

  delete toPop;
  n_nodes--;
}
```

```
void DoublyLinkedList::pop_at(int idx) {
  // TOOD: Implement the pop_at method
  if (idx < 0 || idx >= n_nodes) {
    throw std::out_of_range("Index out of range");
  }

  if (idx == 0) {
    pop_front();
    return;
  }
  if (idx == n_nodes - 1) {
    pop_back();
    return;
  }

  Node *current = head;
  for (int i = 0; i < idx; i++) {
    current = current->next;
  }

  // Update links and delete
  current->prev->next = current->next;
  current->next->prev = current->prev;

  delete current;
  n_nodes--;
}

int DoublyLinkedList::front() {
  // TOOD: Implement the front method
  // replace the line below accordingly
  if (empty()) {
    throw std::runtime_error("List is empty");
  }

  return head->data;
}
```