AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

...Algorytm listy dwukierunkowej z zastosowaniem GitHub...

Autor:
Mateusz Stanek

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest napisanie programu implementującego listę dwukierunkową oraz kontrolowanie jego wersji za pomocą narzędzia git. Lista ma być zaimplementowana za pomocą klasy zawierającej się w innym pliku. Należy także wygenerować automatyczną dokumentację programu za pomocą narzędzia doxygen.

Wynikiem projektu powinna być działająca klasa z implementacją listy, repozytorium git z jej kodem oraz dokumentacja w doxygenie.

2. Analiza problemu

2.1. Lista

List używa się przy okazjach, gdy potrzebny jest kontener potrafiący w szybki i prosty sposób modyfikować swoja wielkość i wewnętrzną strukturę, a szybkość dostępu do samych elementów nie jest aż tak ważna.

Lista jest zbiorem połączonych liniowo ze sobą elementów, gdzie w przypadku tej implementacji, elementy są połączone obustronnie t.zn., każdy element jest połączony z poprzednim i następnym. Dostęp do danego elementu uzyskuje się poprzez enumeracje po kolei wszystkich elementów w liście, aż nie dojdzie się do docelowego. Struktura taka pozwala na łatwe usuwanie i dodawanie elementów - wymaga to tylko zmienienia kilku wskaźników, a nie "przesuwania"całego kontenera.

2.2. Git

Kolejnym konceptem, którym zajmuje się projekt jest narzędzie git. Pozwala ono zarządzać poszczególnymi wersjami projektów. Głównym korzeniem gita jest system commitów, czyli zapisania zmian w pliku w stosunku do commita starszego. To, w połączeniu z jego innymi możliwościami pozwala na tworzenie długich i skomplikowanych osi czasu danych projektów.

2.3. Doxygen

Doxygen jest narzędziem automatycznie generującym dokumentację programu z komentarzy w kodzie źródłowym. Potrafi on generować strony HTML, gdzie można dynamicznie nawigować się miedzy rożnymi częściami kodu oraz pliki LaTeX, które można konwertować na różne, statyczne formaty.

3. Projektowanie

3.1. Implementacja listy

Do zaimplementowania listy zostanie użyty Język `C++` z kompilatorem `g++`. Wersja standardu `C++`to `C++23`. Wersja ta została użyta, ze względu na zawartą w niej funkcję `std::print()`. Jako, że projekt ma być rozdzielony na dwa pliki, zostanie zastosowany `CMake` w celu automatyzacji procesu budowania. `CMake` pozwala na generowanie plików budujących dany projekt, zgodnie z określoną konfiguracją. Oszczędza to programiście, szczególnie przy większych projektach, manualne pisanie Makefileów. Edytorem będzie program Neovim. Jest to terminalowy edytor tekstu z możliwością poszerzenia funkcjonalności przy użyciu wszelkiego rodzaju pluginów. Wybrany został, dlatego że jest on już skonfigurowany na moim komputerze zgodnie z moimi preferencjami.

3.2. Git

Dla ułatwienia pracy, zastosowany został front-end dla gita o nazwie lazygit. Jest to terminalowy program, którego główną zaletą jest łatwa nawigacja przy użyciu klawiatury.Ponadto, jest on lekki i szybki.

3.3. Doxygen

Konfiguracja dla Doxygena jest wygenerowana przy użyciu programu doxywizard, pozwalającego na graficzne zmienianie ustawień. Po wygenerowaniu konfiguracji, doxygen wywoływany jest komendą.

4. Implementacja

4.1. Ogólne informacje o implementacji klasy

Lista jest zaimplementowana jako jeden plik `.hpp`. Nie jest podzielona na plik implementacji oraz nagłówek, ponieważ jest ona szablonem. Deklaracja klasy oraz prywatne elementy wyglądają następująco:

```
1  template<typename T>
2  class DoubleLinkedList {
3  private:
4  struct Node {
5    Node* previous;
6    Node* next;
7    T contents;
8
9    Node(Node* oPrevious, Node* oNext, T oContents):
10     previous { oPrevious },
11     next { oNext },
12     contents { oContents } {}
13
14   ~Node(void) {
15     delete next;
16   }
17 };
18
19 Node* head;
20 Node* tail;
21
22 Node* get_node_at(size_t index) {
23   Node* currentNode { head };
24   for(size_t i {}; i < index; i++) {
25     currentNode = currentNode->next;
26   }
27
28   return currentNode;
29 }
30
31 public:
32 ...
```

Listing 1. Deklaracja szblonu listy

Jak widać w kodzie 1., Klasa jest wrapperem dla structa `Node`. Struct ten ma dwa wskaźniki - `next` dla elementu następnego i `previous` dla elementu poprzedniego.

Jego konstruktor pozwala od razu ustawiać zawartość oraz te wskaźniki Metoda `get_node_at()` jest pomocniczą metodą pozwalającą uzyskać wskaźnik do `Node` o danym indeksie, podążając w przód od wskaźnika head, `index` razy.

Manipulacje strukturą listy odbywają się poprzez szereg metod publicznych. Wiele z nich posiada podobną strukturę. Za przykład jednej z nich można wziąć `prepend()`:

```
1  void prepend(const T& item) {
2    if(head == nullptr) {
3      head = new Node { nullptr, nullptr, item };
4      tail = head;
5      return;
6    }
7
8    head = new Node { nullptr, head, item };
9    head->next->previous = head;
10 }
```

Listing 2. Kod `prepend()`

Metoda z fragmentu nr. 2 ma na celu wstawienie elementu, którego wartość jest zawarta w parametrze `item` na początku listy. Na początku metody, sprawdzane jest czy lista jest pusta - wtedy `head == nullptr`. Jeżeli jest, trzeba stworzyć nowego `Node` na miejscu heada z zawartością będąca parametrem `item`. Jeżeli `head` już istnieje, to też tworzy się nowego `Nodea` na jego miejscu, lecz jako wskaźnik `next` ustawia się adres starego `heada`. Potem we wskaźniku `previous` starego `heada` ustawia się adres nowego heada. Ten zabieg efektywnie sprawił ze stary `head` jest drugi w kolejności listy.

4.2. Ciekawe fragmenty kodu

W metodzie `pop_at()`, mającej na celu usuniecie `Nodea` o danym indeksie, wywoływany jest destruktor danego `Nodea` w taki sposób, aby - ze względu na jego rekursywny charakter - nie usuwać następnych elementów listy. Przy czym ciągłość listy musi być zachowana. Kod metody wygląda następująco:

```
1
2  void pop_at(size_t index) {
3    Node* toPop { get_node_at(index) };
4    if(toPop == head) {
5      rpop();
6      return;
7    } else if(toPop == tail) {
```

```
8       pop();
9       return;
10    }
11
12    toPop->previous->next = toPop->next;
13    toPop->next->previous = toPop->previous;
14    toPop->next = nullptr;
15
16    delete toPop;
17  }
```

Listing 3. Kod `pop_at()`

Jak widać na fragmencie nr. 3, takie zabiegi wymagają niezłej zabawy ze wskaźnikami.

5. Wnioski

- Przy rebaseowaniu, należy zwrócić uwagę, jakie pliki zostaną zmienione.
- Stashe w git są zbawieniem.

Spis rysunków

Spis tabel

Spis listingów