AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm listy dwukierunkowej z zastosowaniem GitHub

Autor:
Mateusz Stanek
Dawid Szołdra

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest stworzenie programu tworzące drzewo BTS działające na stercie oraz kontrolowanie jego wersji za pomocą narzędzia git. Do zadań programu będzie należało: dodawanie elementu, usuwanie elementu, usunięcie całego drzewa, szukanie drogi do wskazanego elementu, wyświetlenie drzewa graficznego, zapis do pliku tekstowego wygenerowanego drzewa.
Program będzie podzielony na 3 pliki: plik main - w którym zostanie zaimplementowane menu sterujące drzewem BTS, plik z drzewem - zawierający klasę drzewa bTS oraz jej implementację oraz plik który będzie odczytywał drzewo oraz zapisywał je do pliku tekstowego pliku tekstwoego.
Wynikiem projektu powinno być działające drzeewo BTS, repozytorium git z kodem oraz dokumentacja w doxygenie.

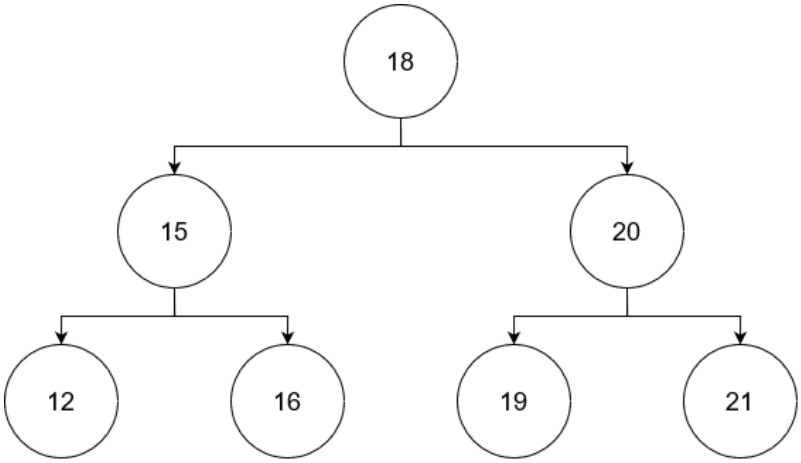
2. Analiza problemu

2.1. Drzewo BST

Drzewo binarne BST jet strukturą przechowującą dane w sposób posortowany. Poszczególne instancje danych są reprezentowane za pomocą węzłów połączonych ze sobą gałęziami. Każdy węzeł posiada dwoje dzieci - lewe oraz prawe. Prawe dziecko jest zawsze większe lub równe od swojego rodzica. Natomiast lewe dziecko jest mniejsze od rodzica i prawego dziecka.

Taka struktura drzewa zapewnia efektywne sortowanie zawartych w nich danych.

Graficzna reprezentacja drzewa BST zaprezentowana na rysunku 2.1



Rys. 2.1. Reprezentacja drzewa BST

Na ryunku 2.1 węzły są reprezentowane przez okręgi a gałęzie przez strzałki.

2.2. Git

Kolejnym konceptem, którym zajmuje się projekt jest narzędzie git[1]. 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.

Użycie gita można zademonstrować na prostym przykładzie. Tworzymy katalog a w nim repozytorium, uzywając komendy `git init`, jak widać na rys. 2.2.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/mattys/skrypty-i-syfy/studia/rok2/progr
amowanie-zaawansowane/p1/git-test/.git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % ls
drwxr-xr-x - mattys  6 Nov 15:54 .git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test %
```

Rys. 2.2. Puste repozytorium git

Stwórzmy jakiś plik i dodajmy go do repozytorium. Plik można dodać do repozytorium komendą `git add`

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit1" > plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git add plik.txt
```

Rys. 2.3. Stworznie pliku w repozytorium

Następnie należy scommitować zmiany.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "utworzenie pl
ik.txt"
[master (root-commit) bb3b898] utworzenie plik.txt
 1 file changed, 1 insertion(+)
 create mode 100644 plik.txt
```

Rys. 2.4. Commit nr. 1

Na rysunku 2.4 użyta komenda `git commit` commituje wszystkie dodane pliki (`-a`) z jakimś komunikatem (`-m`).

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit2" >> plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "uzupelnienie
plik.txt"
[master 40694c2] uzupelnienie plik.txt
 1 file changed, 1 insertion(+)
```

Rys. 2.5. Commit nr. 2

Na rys. 2.5, został utworzony kolejny commit, dodajacy zmiany do `plik.txt`.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git log
commit 40694c2e73758368445cb817f4524ee5c5afbece (HEAD → master)
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:26:07 2024 +0100

    uzupelnienie plik.txt

commit bb3b898df760395b5763575e204c3c43b513d2f6
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:12:31 2024 +0100

    utworzenie plik.txt
```

Rys. 2.6. Log gita

Jak na rys. 2.6 jest pokazane, używając komendy `git log`, można wyświetlić log commitów w repozytorium.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git checkout bb3b898df760395b
5763575e204c3c43b513d2f6
Note: switching to 'bb3b898df760395b5763575e204c3c43b513d2f6'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at bb3b898 utworzenie plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo plik.txt
plik.txt
```

Rys. 2.7. Demonstracja checkout

Jak widać na rys. 2.7, komenda `git checkout`, pozwala na przejście repozytorium w inny stan, w tym przypadku przechodzi się do commita o danym ID, pokazanym na rys. 2.6. Jako, że jest to pierwszy commit, nie ma w nim zmian z drugiego.

2.3. Doxygen

Doxygen[2] 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 drzewa BTS

Do zaimplementowania drzewa BTS zostanie użyty Język `C++` z kompilatorem `g++` oraz `MVSC`. 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. Plik konfiguracyjny `CMakeLists.txt` może wyglądać jak na rysunku

```
1  cmake_minimum_required(VERSION 3.5)
2
3  set(PROJECT_NAME proj1)
4
5  project(${PROJECT_NAME} VERSION 0.1 LANGUAGES CXX)
6
7  set(CMAKE_CXX_STANDARD 23)
8  set(CMAKE_CXX_STANDARD_REQUIRED ON)
9  set(CMAKE_EXPORT_COMPILE_COMMANDS True)
10 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/out)
11
12 add_subdirectory(src)
13
```

Listing 1. Plik konfiguracyjny CMake

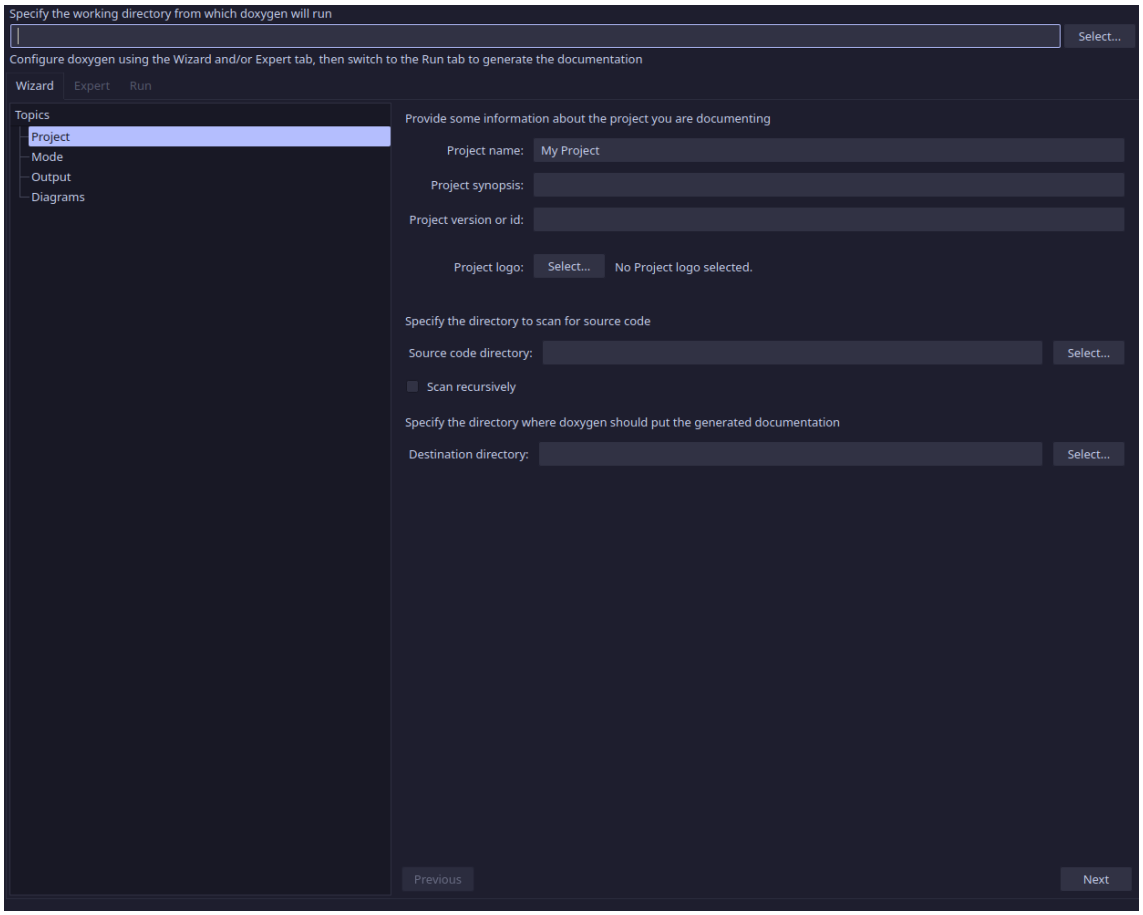
Edytorem będzie program Neovim oraz Visual Studio 2022. Jest to terminalowy edytor tekstu z możliwością poszerzenia funkcjonalności przy użyciu wszelkiego rodzaju pluginów. Wybrany został, dlatego że jest on już skonfigurowany na moim komputerze zgodnie z moimi preferencjami.

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, pokazany na rys. 3.1, pozwalającego na graficzne zmienianie ustawień. Po wygenerowaniu konfiguracji, Doxygen wywoływany jest przy użyciu komendy.



Rys. 3.1. Interfejs programu doxywizard

4. Implementacja

4.1. Ogólne informacje o implementacji klasy

Drzewo jest zaimplementowane jako jeden plik `.hpp`. Nie jest podzielone na plik implementacji oraz nagłówek, ponieważ jest ono szablonem. Deklaracja klasy oraz prywatne elementy wyglądają następująco:

```
1  template <typename T>
2  class BSTTree {
3    private:
4    struct Tree {
5      T contents;
6      Tree* parent;
7      Tree* left;
8      Tree* right;
9
10     Tree(T _contents, Tree* _parent = nullptr, Tree* _left =
   nullptr, Tree* _right = nullptr):
11     contents { _contents },
12     parent { _parent },
13     left { _left },
14     right { _right } {}
15
16     ~Tree() {
17       delete left;
18       delete right;
19     }
20   };
21
22   Tree* root;
23
24   void recursive_add(const T& element, Tree*& node, Tree*
   parentNode = nullptr) {
25     if(node == nullptr) {
26       node = new Tree(element);
27
28       if(parentNode == nullptr) {
29         return;
30       }
31
32       node->parent = parentNode;
33
34       if(node->contents >= parentNode->contents) {
35         parentNode->right = node;
```

```
36        } else {
37          parentNode->left = node;
38        }
39
40        return;
41      }
42
43      if(element >= node->contents) {
44        recursive_add(element, node->right, node);
45      } else {
46        recursive_add(element, node->left, node);
47      }
48    }
49
50    void preorder_traverse_recursive(Tree* node, std::vector<Tree
    *>& traversedTrees) {
51      if(node == nullptr) {
52        return;
53      }
54
55      traversedTrees.push_back(node);
56
57      preorder_traverse_recursive(node->left, traversedTrees);
58      preorder_traverse_recursive(node->right, traversedTrees);
59    }
60
61    void inorder_traverse_recursive(Tree* node, std::vector<Tree*>&
     traversedTrees) {
62      if(node == nullptr) {
63        return;
64      }
65
66      inorder_traverse_recursive(node->left, traversedTrees);
67
68      traversedTrees.push_back(node);
69
70      inorder_traverse_recursive(node->right, traversedTrees);
71    }
72
73    void postorder_traverse_recursive(Tree* node, std::vector<Tree
    *>& traversedTrees) {
74      if(node == nullptr) {
75        return;
76      }
77
```

```
78      postorder_traverse_recursive(node->left, traversedTrees);
79      postorder_traverse_recursive(node->right, traversedTrees);
80
81      traversedTrees.push_back(node);
82    }
83    public:
84    ...
```

Listing 2. Deklaracja drzewa BST

Jak widać w kodzie 2., Klasa jest wrapperem dla structa `Tree`. Struct ten ma trzy wskaźniki - `left` dla elementu lewego, `right` dla elementu prawego, `parent` dla elementu będącego rodzicem oraz zawartość `contents` nie będącą wskaźnikiem. W linii 10 znajduje się konstruktor, który ustawia wartości wskaźników rodzica, dziecka lewego oraz prawego na null. Od linii 11 do 14 jest konstruktor który pozwala ustawić wskaźniki i wymaga ustawienia zawartości. Od linii 16 do 18 znajduje się destruktor drzewa. W destruktorze węzeł lewy i prawy jest usuwany.

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 3. Kod `prepend()`

Metoda z fragmentu nr. 3 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 4. Kod `pop_at()`

Jak widać na fragmencie nr. 4, takie zabiegi wymagają niezłej zabawy ze wskaźnikami.

5. Wnioski

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

Bibliografia

[1] *Strona Gita*. URL: https://git-scm.com/.

[2] *Strona Doxygena*. URL: https://www.doxygen.nl/.

Spis rysunków

Spis tabel

Spis listingów