UNIVERSITAS ESA UNGGUL
CSF101 ALGORITMA DAN PEMROGRAMAN KJ1001 7174

ALGORITMA STACK DALAM C++ - IMPLEMENTASI DAN ANALISIS STRUKTUR DATA LIFO DALAM PEMROGRAMAN MODERN

TUGAS KELOMPOK 7

Dosen Pengampu:
7174 - Ir. Sawali Wahyu, S.Kom., M.Kom

Disusun Oleh Kelompok 7:

1. **Muhamad Akbar Fadilah - 20200801269**
2. **Christian Niko Saputra - 20240801295**
3. **Denis Prastya Putra - 20240801319**
4. **Davina Tri Febriyanti - 20240801361**
5. **Arva Raihan Javier - 20240801344**

PROGRAM STUDI SISTEM INFORMASI
FAKULTAS ILMU KOMPUTER
UNIVERSITAS ESA UNGGUL
TAHUN 2025

DAFTAR ISI

ABSTRAK

Struktur data merupakan komponen fundamental dalam ilmu komputer yang mempengaruhi efisiensi dan kinerja program. Di antara berbagai struktur data yang ada, Stack memiliki peran unik dengan prinsip Last In, First Out (LIFO). Makalah ini mengkaji secara mendalam tentang implementasi Stack dalam bahasa pemrograman C++, termasuk analisis komprehensif terhadap metode-metode yang tersedia, kasus penggunaan praktis, serta evaluasi kinerja dalam berbagai skenario. Penelitian ini menggunakan pendekatan analitis dengan fokus pada implementasi Stack menggunakan Standard Template Library (STL) C++. Hasil analisis menunjukkan bahwa Stack memiliki efisiensi tinggi untuk operasi LIFO dengan kompleksitas waktu O(1) untuk operasi dasar, menjadikannya pilihan ideal untuk berbagai aplikasi seperti manajemen memori, evaluasi ekspresi, dan implementasi undo-redo dalam software.

Kata Kunci: *Stack, LIFO, C++, STL, Struktur Data, Algoritma*

BAB I – PENDAHULUAN

1. Latar Belakang

Dalam era komputasi modern, efisiensi pengelolaan data menjadi faktor kritis dalam pengembangan software. Stack, sebagai salah satu struktur data fundamental, memegang peranan penting dalam berbagai aplikasi pemrograman. Prinsip Last In, First Out (LIFO) yang dimiliki Stack membuatnya ideal untuk menangani berbagai kasus penggunaan seperti manajemen memori program, evaluasi ekspresi matematis, dan implementasi fitur undo-redo dalam aplikasi.

Pemahaman mendalam tentang Stack tidak hanya penting dari perspektif akademis tetapi juga memiliki implikasi praktis yang signifikan dalam pengembangan software. Dalam konteks bahasa pemrograman C++, Stack telah diimplementasikan sebagai bagian dari Standard Template Library (STL), menyediakan interface yang terstandarisasi dan efisien untuk manipulasi data.

2. Rumusan Masalah

1. Bagaimana implementasi Stack dalam C++ dapat dioptimalkan untuk berbagai kasus penggunaan?
2. Apa saja metode-metode yang tersedia dalam Stack C++ dan bagaimana efektivitasnya?
3. Bagaimana performa Stack dibandingkan dengan struktur data lain dalam konteks operasi LIFO?
4. Apa tantangan dan keterbatasan penggunaan Stack dalam pengembangan aplikasi modern?

3. Tujuan Penelitian

1. Menganalisis implementasi Stack dalam C++ secara komprehensif.
2. Mengevaluasi efektivitas metode-metode Stack dalam berbagai skenario penggunaan.
3. Mengidentifikasi best practices dalam penggunaan Stack untuk pengembangan aplikasi.
4. Memberikan rekomendasi untuk optimasi penggunaan Stack dalam konteks pemrograman modern.

4. Manfaat Penelitian

1. Memberikan pemahaman mendalam tentang Stack bagi pengembang software.
2. Menyediakan panduan praktis untuk implementasi Stack dalam C++.
3. Membantu optimasi kinerja aplikasi melalui penggunaan Stack yang efektif.
4. Berkontribusi pada pengembangan ilmu pengetahuan dalam bidang struktur data.

BAB II – LANDASAN TEORI

1. Definisi dan Konsep Dasar Stack

Stack adalah struktur data linier yang fundamental dalam ilmu komputer, yang mengorganisir dan menyimpan data menggunakan prinsip Last-In-First-Out (LIFO). Dalam prinsip LIFO ini, elemen atau data yang terakhir dimasukkan (pushed) ke dalam stack akan menjadi elemen pertama yang dikeluarkan (popped) dari stack. Struktur data ini memiliki karakteristik unik di mana operasi penambahan dan penghapusan elemen hanya dapat dilakukan pada satu ujung stack yang disebut "top" atau puncak, yang membuat pengelolaan datanya menjadi sangat terstruktur dan prediktif.

Dalam implementasi praktisnya, Stack dapat dianalogikan dengan berbagai contoh dalam kehidupan sehari-hari, seperti tumpukan piring di rak piring, tumpukan buku di meja, atau tumpukan kartu di meja permainan. Ketika seseorang menambahkan piring baru ke tumpukan, piring tersebut akan diletakkan di atas piring-piring yang sudah ada sebelumnya, dan ketika seseorang ingin mengambil piring, mereka akan mengambil piring yang berada di posisi paling atas terlebih dahulu. Analogi ini secara sempurna menggambarkan bagaimana Stack beroperasi dalam konteks pemrograman, di mana setiap operasi dieksekusi dengan urutan yang konsisten dan dapat diprediksi.

Dalam konteks pemrograman komputer, Stack memiliki peran vital dalam berbagai aplikasi dan proses komputasi. Stack digunakan secara luas dalam manajemen memori program, di mana setiap pemanggilan fungsi dan variabel lokalnya disimpan dalam stack memory. Stack juga berperan penting dalam implementasi fitur undo-redo pada aplikasi, evaluasi ekspresi matematis (terutama dalam notasi postfix), pelacakan pemanggilan fungsi (call stack), dan dalam algoritma pencarian seperti Depth-First Search (DFS). Keunggulan Stack terletak pada kemampuannya untuk mengelola data dengan cara yang terorganisir dan efisien, dengan kompleksitas waktu O(1) untuk operasi dasar seperti push dan pop, membuatnya menjadi pilihan ideal untuk aplikasi yang memerlukan pengelolaan data dengan prinsip LIFO.

2. Prinsip LIFO

Last-In-First-Out (LIFO) adalah prinsip fundamental yang mendefinisikan cara kerja dan karakteristik utama dari struktur data Stack. Prinsip ini menetapkan bahwa elemen yang terakhir dimasukkan ke dalam Stack akan menjadi elemen pertama yang dikeluarkan, menciptakan urutan pemrosesan data yang sangat spesifik dan terkontrol. Dalam implementasinya, prinsip LIFO membatasi akses dan manipulasi data hanya pada satu titik dalam struktur data, yaitu bagian puncak (top) dari Stack. Ketika sebuah elemen baru ditambahkan (push operation), elemen tersebut secara otomatis menjadi elemen puncak dan menutupi akses ke elemen-elemen yang telah ada sebelumnya di dalam Stack.

Hal ini menciptakan sebuah mekanisme penyimpanan data yang sangat terstruktur dan deterministik, di mana urutan penghapusan elemen akan selalu merupakan kebalikan dari urutan penambahannya.

Implementasi prinsip LIFO dalam Stack memiliki implikasi penting pada cara data dikelola dan diakses. Karakteristik ini membuat Stack menjadi struktur data yang ideal untuk skenario di mana urutan pemrosesan terbalik diperlukan, seperti dalam pelacakan history program atau pembatalan operasi (undo operations). Keterbatasan akses yang hanya dimungkinkan pada elemen puncak memastikan integritas data dan mencegah manipulasi yang tidak diinginkan pada elemen-elemen di dalam Stack. Meskipun hal ini dapat dilihat sebagai pembatasan, justru karakteristik inilah yang membuat Stack sangat efisien dalam operasi dasarnya, dengan kompleksitas waktu O(1) untuk operasi push dan pop, karena tidak diperlukan pencarian atau pengaturan ulang elemen-elemen lain dalam struktur data. Keuntungan dari pendekatan ini adalah prediktabilitas yang tinggi dalam perilaku program dan kemudahan dalam melacak alur data, meskipun dengan konsekuensi tidak tersedianya akses acak ke elemen-elemen yang berada di tengah Stack.

Prinsip LIFO merupakan karakteristik fundamental dari Stack yang membedakannya dari struktur data lain. Dalam implementasinya:

1. Elemen baru selalu ditambahkan ke "puncak" Stack.
2. Penghapusan elemen hanya dapat dilakukan dari puncak Stack.
3. Akses langsung ke elemen selain elemen puncak tidak dimungkinkan.

3. Operasi Fundamental Stack

Operasi fundamental Stack merupakan sekumpulan operasi dasar yang menjadi fondasi dalam pengelolaan dan manipulasi data dalam struktur Stack. Operasi-operasi ini dirancang dengan mempertimbangkan prinsip LIFO (Last-In-First-Out) sebagai karakteristik utama Stack, memastikan bahwa setiap manipulasi data dilakukan secara konsisten dan efisien. Setiap operasi fundamental memiliki peran spesifik yang tidak dapat dipisahkan dalam memastikan integritas dan fungsionalitas Stack sebagai struktur data.

Dalam konteks pemrograman, operasi fundamental Stack merepresentasikan interface standar yang harus disediakan oleh setiap implementasi Stack, terlepas dari bahasa pemrograman atau platform yang digunakan. Operasi-operasi ini memungkinkan programmer untuk melakukan manipulasi data secara terstruktur dan prediktif, dengan kompleksitas waktu yang konstan O(1) untuk sebagian besar operasi. Setiap operasi fundamental dirancang untuk menjaga integritas data dan konsistensi struktur Stack, sambil memastikan efisiensi dalam penggunaan memori dan waktu eksekusi.

Stack memiliki beberapa operasi dasar yang mendefinisikan perilakunya:

1. Push: Menambahkan elemen ke puncak Stack

2. Pop: Menghapus elemen dari puncak Stack
3. Top/Peek: Mengakses elemen di puncak Stack tanpa menghapusnya
4. IsEmpty: Memeriksa apakah Stack kosong
5. Size: Mendapatkan jumlah elemen dalam Stack

4. Kompleksitas Waktu Operasi Stack

Stack adalah salah satu struktur data yang menerapkan prinsip LIFO (Last In, First Out), di mana elemen yang terakhir dimasukkan adalah elemen pertama yang akan dikeluarkan. Karena implementasi stack biasanya dilakukan menggunakan array atau linked list sederhana, sebagian besar operasi dasar stack memiliki kompleksitas waktu konstan, yaitu O(1). Operasi-operasi ini melibatkan akses atau manipulasi elemen pada posisi teratas stack, yang dapat dilakukan tanpa perlu traversal atau pencarian elemen lainnya. Hal ini membuat stack sangat efisien untuk kasus penggunaan yang memerlukan penanganan data secara terstruktur dalam urutan LIFO.

Secara rinci, operasi seperti push (menambahkan elemen ke atas stack) dan pop (menghapus elemen dari atas stack) hanya membutuhkan sedikit langkah karena mereka hanya berinteraksi dengan elemen paling atas. Demikian pula, operasi top yang mengembalikan elemen teratas tanpa menghapusnya, isEmpty yang memeriksa apakah stack kosong, dan size yang menghitung jumlah elemen di dalam stack semuanya hanya memerlukan akses langsung ke atribut atau pointer yang sudah tersedia. Efisiensi ini membuat stack menjadi pilihan populer dalam berbagai algoritma, seperti pemrosesan ekspresi matematika, penelusuran graf menggunakan metode DFS, atau menjaga konteks eksekusi dalam pemrograman rekursif.

Analisis kompleksitas waktu operasi Stack:

1. Push: O(1)
2. Pop: O(1)
3. Top: O(1)
4. IsEmpty: O(1)
5. Size: O(1)

BAB III – IMPLEMENTASI DAN PEMBAHASAN

1. Implementasi Stack dalam C++

Dalam C++, Stack dapat diimplementasikan melalui beberapa pendekatan yang berbeda, masing-masing dengan karakteristik dan use case yang spesifik. Implementasi Stack dalam C++ dapat dilakukan baik menggunakan Standard Template Library (STL) maupun melalui implementasi manual.

1. Header dan Namespace

#include <stack>

using namespace std;

Kode #include <stack> digunakan untuk menyertakan header file stack dalam program C++. Header ini menyediakan implementasi struktur data stack bawaan di pustaka standar C++ (STL). Dengan menggunakan header ini, Anda dapat memanfaatkan semua fungsi dan metode bawaan yang disediakan untuk stack, seperti push(), pop(), top(), empty(), dan size(). Stack yang disediakan oleh STL adalah generik, artinya Anda dapat membuat stack untuk berbagai jenis data, seperti int, float, string, atau tipe data kustom.

Pernyataan using namespace std; digunakan untuk menghindari penulisan prefiks std:: sebelum nama elemen yang berasal dari pustaka standar C++. Dengan kata lain, jika Anda menyertakan using namespace std;, Anda dapat langsung menggunakan nama seperti stack alih-alih harus menuliskan std::stack. Meskipun pendekatan ini mempermudah penulisan kode, sebaiknya berhati-hati dalam penggunaannya, terutama pada program besar, untuk menghindari potensi konflik nama (namespace collision).

2. Deklarasi dan Inisialisasi

// Stack of integers

stack<int> number_stack;

// Stack of strings

stack<string> text_stack;

// Stack of custom objects

stack<CustomClass> object_stack;

Deklarasi dan inisialisasi stack di C++ dilakukan dengan menggunakan pustaka standar (STL) yang menyediakan struktur data generik untuk berbagai tipe data. Sebagai contoh, stack<int> number_stack; mendeklarasikan sebuah stack yang menyimpan elemen bertipe integer, cocok untuk kebutuhan pengelolaan data numerik. Selain itu, stack<string> text_stack; mendefinisikan stack untuk menyimpan elemen bertipe string, yang ideal untuk memproses teks atau kata-kata. Untuk tipe data kustom, seperti CustomClass, deklarasi stack<CustomClass> object_stack; memungkinkan pengelolaan elemen berupa objek dengan properti tertentu, menjadikannya fleksibel untuk aplikasi yang kompleks.

Semua stack yang dideklarasikan ini bersifat generik, sehingga dapat menyimpan data sesuai tipe yang ditentukan tanpa memerlukan implementasi tambahan. Setelah inisialisasi, stack dalam keadaan kosong hingga elemen ditambahkan menggunakan operasi seperti push(). Fleksibilitas ini menjadikan stack sebagai struktur data yang serbaguna dalam pengelolaan data bertumpuk sesuai prinsip LIFO (Last In, First Out).

3. **Implementasi Operasi Dasar**

```
// Push operation
void pushElement(stack<int>& st, int value) {
  st.push(value);
  cout << "Pushed: " << value << endl;
}

// Pop operation
void popElement(stack<int>& st) {
  if (!st.empty()) {
    cout << "Popped: " << st.top() << endl;
    st.pop();
  } else {
    cout << "Stack is empty" << endl;
  }
}
```

```
// Check top element
int checkTop(stack<int>& st) {
  if (!st.empty()) {
    return st.top();
  }
  throw runtime_error("Stack is empty");
}
```

Operasi dasar pada stack meliputi push, pop, dan memeriksa elemen teratas (check top element), yang diimplementasikan menggunakan metode bawaan STL. Operasi push direalisasikan melalui fungsi pushElement, yang menambahkan elemen baru ke atas stack menggunakan metode push() dan mencetak pesan konfirmasi seperti "Pushed: 10". Untuk menghapus elemen teratas, fungsi popElement memanfaatkan metode pop(). Sebelum menghapus, fungsi ini memeriksa apakah stack kosong menggunakan empty() untuk menghindari error. Jika kosong, fungsi mencetak pesan "Stack is empty", sementara jika ada elemen, fungsi mencetak elemen yang dihapus.

Selanjutnya, untuk memeriksa elemen teratas tanpa menghapusnya, fungsi checkTop menggunakan metode top(). Fungsi ini terlebih dahulu memeriksa apakah stack kosong; jika tidak, elemen teratas dikembalikan, tetapi jika kosong, fungsi akan melemparkan pengecualian runtime_error dengan pesan "Stack is empty". Implementasi ini memastikan stack dapat dikelola dengan aman dan efisien sesuai prinsip LIFO (Last In, First Out).

2. Kasus Penggunaan Praktis

1. Implementasi Undo-Redo

```
class TextEditor {
private:
  stack<string> undoStack;
  stack<string> redoStack;
  string currentText;

public:
```

```
  void type(string text) {
    undoStack.push(currentText);
    currentText = text;
    // Clear redo stack when new action is performed
    while (!redoStack.empty()) redoStack.pop();
  }

  void undo() {
    if (!undoStack.empty()) {
      redoStack.push(currentText);
      currentText = undoStack.top();
      undoStack.pop();
    }
  }

  void redo() {
    if (!redoStack.empty()) {
      undoStack.push(currentText);
      currentText = redoStack.top();
      redoStack.pop();
    }
  }
};
```

Dalam kode di atas, kita melihat implementasi sistem undo dan redo untuk editor teks sederhana yang mengelola perubahan teks dengan menggunakan dua stack: undoStack dan redoStack. Stack pertama menyimpan teks sebelumnya untuk memungkinkan pengguna mengembalikan perubahan (undo), sementara stack kedua menyimpan teks yang telah di-undo, memungkinkan pengguna untuk mengulang perubahan tersebut (redo).

Pada fungsi type, setiap kali pengguna mengetik teks baru, teks sebelumnya disimpan ke dalam undoStack dan teks baru disalin ke variabel currentText. Selain itu, setiap kali tindakan baru dilakukan, stack redoStack akan dibersihkan untuk memastikan tidak ada tindakan yang dapat diulang sebelum aksi baru. Fungsi undo memeriksa apakah undoStack tidak kosong. Jika tidak kosong, teks saat ini disalin ke redoStack dan kemudian diubah ke teks yang ada di undoStack. Fungsi redo melakukan hal yang hampir sama, dengan memindahkan teks saat ini ke undoStack dan mengambil teks dari redoStack. Implementasi ini memastikan bahwa setiap perubahan dapat diulang kembali atau dibatalkan sesuai urutan yang diinginkan, menciptakan pengalaman pengeditan yang fleksibel dan efisien.

2. **Evaluasi Ekspresi Matematis**

```
bool isOperator(char c) {
  return (c == '+' || c == '-' || c == '*' || c == '/');
}

int evaluatePostfix(string expression) {
  stack<int> st;

  for (char c : expression) {
    if (isdigit(c)) {
      st.push(c - '0');
    }
    else if (isOperator(c)) {
      int val2 = st.top(); st.pop();
      int val1 = st.top(); st.pop();

      switch(c) {
        case '+': st.push(val1 + val2); break;
        case '-': st.push(val1 - val2); break;
        case '*': st.push(val1 * val2); break;
        case '/': st.push(val1 / val2); break;
```

```
        }
      }
    }
    return st.top();
}
```

Kode di atas menunjukkan implementasi untuk mengevaluasi ekspresi matematis dalam notasi postfix (Reverse Polish Notation, RPN) menggunakan stack. Fungsi isOperator memeriksa apakah karakter yang diberikan merupakan operator aritmatika dasar, seperti penjumlahan, pengurangan, perkalian, atau pembagian. Operator-operator ini digunakan untuk menentukan tindakan yang akan dilakukan pada dua nilai yang diambil dari stack.

Fungsi evaluatePostfix melakukan evaluasi ekspresi postfix dengan memproses setiap karakter dalam ekspresi satu per satu. Jika karakter tersebut adalah digit, maka nilai numerik tersebut dimasukkan ke dalam stack. Jika karakter tersebut adalah operator, dua nilai teratas dari stack diambil dan operasi yang sesuai (penjumlahan, pengurangan, perkalian, atau pembagian) dilakukan. Hasil operasi kemudian dimasukkan kembali ke dalam stack. Setelah seluruh ekspresi diproses, nilai terakhir yang tersisa di stack adalah hasil akhir dari evaluasi ekspresi postfix. Implementasi ini efektif untuk mengevaluasi ekspresi matematis dengan kompleksitas waktu O(n), di mana n adalah jumlah karakter dalam ekspresi.

3. **Analisis Kinerja**

    1. **Penggunaan Memori**
       Stack memiliki penggunaan memori yang efisien karena:
        i. Alokasi memori statis dan deterministik
        ii. Overhead minimal untuk metadata
        iii. Tidak ada fragmentasi memori

    2. **Performa Operasi**

       Hasil pengujian menunjukkan:

        i. Operasi push dan pop konsisten O(1)
        ii. Penggunaan memori linear dengan jumlah elemen
        iii. Kinerja optimal untuk operasi LIFO

    3. **Perbandingan dengan Struktur Data Lain**

Operasi	Stack	Queue	Vector
Insert	O(1)	O(1)	O(1)*
Delete	O(1)	O(1)	O(n)
Access	O(1)	O(n)	O(1)
Search	O(n)	O(n)	O(n)

BAB IV – BEST PRACTICES DAN OPTIMASI

1. **Manajemen Memori**
    1. Preallocation untuk kasus penggunaan dengan ukuran yang dapat diprediksi
    2. Clear stack ketika tidak digunakan lagi
    3. Penggunaan move semantics untuk objek besar
2. **Exception Handling**

```
template<typename T>
class SafeStack {
private:
  stack<T> st;

public:
  void safePush(const T& value) {
    try {
      st.push(value);
    } catch (const std::bad_alloc& e) {
      throw runtime_error("Memory allocation failed during push operation");
    }
  }

  T safePop() {
    if (st.empty()) {
      throw runtime_error("Cannot pop from empty stack");
    }
    T value = st.top();
    st.pop();
    return value;
  }
};
```

Kode di atas menunjukkan implementasi stack yang aman (SafeStack) dengan penanganan pengecualian (exception handling). Kelas SafeStack adalah template kelas generik yang memungkinkan pembuatan stack untuk berbagai tipe data. Pada kelas ini, dua operasi dasar stack, yaitu safePush dan safePop, diimplementasikan dengan penanganan pengecualian untuk memastikan operasi yang aman, terutama dalam situasi error.

Fungsi safePush bertanggung jawab untuk menambahkan elemen ke dalam stack. Ketika operasi push dilakukan, pengecualian std::bad_alloc akan ditangani jika terjadi kegagalan dalam alokasi memori, dan kemudian melemparkan pengecualian baru dengan pesan "Memory allocation failed during push operation". Sementara itu, fungsi safePop mengeluarkan elemen dari stack. Sebelum melakukan operasi pop, fungsi ini memeriksa apakah stack kosong menggunakan empty(). Jika stack kosong, fungsi akan melemparkan pengecualian runtime_error dengan pesan "Cannot pop from empty stack". Dengan pendekatan ini, kesalahan selama operasi stack dapat ditangani dengan lebih aman dan efisien, mencegah program dari crash atau perilaku tak terduga.

BAB V – HASIL DAN PEMBAHASAN

1. **Kelebihan Stack**
    1. Implementasi sederhana dan straightforward
    2. Kompleksitas waktu O(1) untuk operasi dasar
    3. Penggunaan memori efisien
    4. Ideal untuk operasi LIFO
    5. Dukungan STL C++ yang komprehensif
2. **Keterbatasan**
    1. Akses terbatas pada elemen puncak
    2. Tidak mendukung iterasi langsung
    3. Kapasitas terbatas pada ukuran memori yang tersedia
    4. Tidak cocok untuk akses acak
3. **Rekomendasi Penggunaan**
    1. Manajemen state dalam aplikasi
    2. Implementasi undo-redo
    3. Evaluasi ekspresi dan parsing
    4. Tracking history operasi
    5. Manajemen memori program

BAB VI – KESIMPULAN DAN SARAN

1. Kesimpulan

Stack adalah struktur data yang sangat efisien dalam pengelolaan elemen dengan prinsip Last In, First Out (LIFO). Semua operasi dasar stack, seperti push, pop, top, isEmpty, dan size, memiliki kompleksitas waktu O(1), yang berarti bahwa waktu eksekusi operasi tersebut tidak bergantung pada jumlah elemen yang ada dalam stack. Hal ini membuat stack sangat cocok digunakan dalam berbagai aplikasi yang memerlukan akses cepat ke elemen terakhir yang dimasukkan, seperti dalam algoritma rekursi, traversal graf, atau pemrosesan ekspresi matematika.

Implementasi stack dalam C++ menggunakan pustaka standar (STL) sangat memudahkan pengguna karena menyediakan antarmuka yang sederhana dan efisien. Dengan menggunakan kelas stack yang disediakan oleh STL, pengembang dapat langsung menggunakan berbagai metode seperti push(), pop(), dan top() tanpa perlu mengimplementasikan struktur data dari awal. Hal ini menghemat waktu pengembangan dan mengurangi potensi kesalahan implementasi. Dengan kemampuan untuk menyimpan berbagai tipe data melalui template, stack di C++ menjadi struktur yang fleksibel dan kuat.

Meskipun stack sangat efisien untuk pengelolaan data LIFO, struktur data ini memiliki keterbatasan, terutama dalam hal akses data. Karena stack hanya memungkinkan akses pada elemen teratas, pengelolaan elemen lainnya menjadi tidak langsung. Namun, meskipun memiliki keterbatasan ini, stack tetap menjadi pilihan optimal untuk kasus penggunaan yang membutuhkan pengelolaan data secara terstruktur, seperti dalam aplikasi undo-redo, pemrosesan ekspresi aritmatika, atau dalam algoritma yang membutuhkan penyimpanan sementara data selama proses rekursif. Dengan segala keunggulannya, stack tetap menjadi alat yang sangat berguna dalam pengembangan perangkat lunak..

2. Saran

1. Pengembangan lebih lanjut untuk mendukung iterasi
2. Implementasi thread-safe stack untuk aplikasi konkuren
3. Optimasi penggunaan memori untuk aplikasi berskala besar
4. Integrasi dengan struktur data lain untuk fungsionalitas yang lebih kompleks

DAFTAR PUSTAKA

Stroustrup, B. (2013). The C++ Programming Language, 4th Edition. Addison-Wesley Professional.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms, 3rd Edition. MIT Press.

Meyers, S. (2014). Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media.

Presentasi "ALGORITMA & PEMROGRAMAN Stack Algorithm C++" oleh Kelompok 7

C++ Standard Template Library Documentation

Josuttis, N. M. (2012). The C++ Standard Library: A Tutorial and Reference, 2nd Edition. Addison-Wesley Professional.