Тема 2. ПОДДЕРЖКА ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Механизм исключительных ситуаций предоставляет пользователю возможность контроля хода выполнения программы и нейтрализации возможных ошибок. Очень часто исключительная ситуация — это не ошибка, а просто исчерпание какого-то ресурса, например, доступной памяти или времени ожидания сигнала или даже один из предусмотренных вариантов завершения процесса. Механизм используется, если ситуация не может быть разрешена в той точке, где была выявлена, и требует перехода на более высокий уровень иерархии вызовов функций, с завершением активных функций и освобождением ресурсов.

Для задействования механизма особых ситуаций в программе нужно проделать следующее:

— обнаружив в программе место, где особая ситуация может возникнуть, надо придумать для неё уникальное название, например *My_Error*, и объявить соответствующий класс ошибок, возможно, пустой:

```
class My_Error {};
```

— в точке программы, в которой обнаружена особая ситуация, поместить утверждение

```
throw My_Error( );
```

Это утверждение создаёт объект класса *My_Error*. Если в объекте предусмотрены поля для данных, через них можно передать информацию обработчику ошибок: аргумент утверждения *throw* — это конструктор объекта;

— точку вызова функции, которая может создать исключения, нужно поместить в блок контроля, за которым следуют обработчики особых ситуаций:

```
try{ //Начало блока контроля.
      Алгоритм, использующий функцию,
      которая может создать особую ситуацию
     (содержит утверждения throw My_Error( ));
}
catch (My_Error)
{ Обработка особой ситуации }
```

Предложение *catch* размещается на том уровне вложенности функций, где обработка ситуации *My_Error* возможна. Если нужно обрабатывать несколько различных ошибок, после блока *try* последовательно размещаются

соответствующие обработчики. При возникновении любой особой ситуации в блоке *try* его работа прерывается: происходит принудительный выход из всей цепочки вложенных функций, активных в точке особой ситуации, и вызов деструкторов для всех созданных при этом объектов, как это происходит при выходе из блока (области видимости). Этот процесс называется раскруткой стека: стек возвращается в состояние, в котором он был в момент входа в блок *try*.

Далее просматриваются блоки *catch* в том порядке, в каком они объявлены. Как только обнаруживается блок обработки ошибки нужного типа, управление передаётся ему. Остальные блоки *catch* не используются.

Если же выполнение блока *try* завершилось успешно, все блоки catch после него игнорируются.

Если для некоторого типа ошибки не обнаружено соответствующего блока *catch*, программа завершается аварийно. Чтобы этого избежать, последним в цепочке можно разместить блок *catch*(…), перехватывающий ошибки любого типа.

Как только подходящий блок *catch* будет вызван, особая ситуация будет считаться обработанной, даже если этот блок пуст. Однако чаще всего в него помещают выдачу на экран или в специальный файл (журнал) содержательного сообщения об ошибке. Возможно также одно из следующих действий:

— устранение причины ошибки (уменьшение запроса на выделение памяти, отказ от обработки несуществующего или испорченного файла и т. п.);

— аварийное завершение программы (вызов *abort*( ));

— возбуждение особой ситуации другого типа (вызов *throw* с соответствующим аргументом);

— перевозбуждение особой ситуации для передачи её на следующий уровень иерархии вызовов функций (вызов *throw* без аргумента).

Подробнее об особых ситуациях и их обработке *см*. [3, с. 222–230], [15, с. 232–256],.

Правильный выбор уровня для размещения блока контроля позволяет сделать программу безопасной в смысле исключений (*см*. [12, с. 105–174]). Так, в учебном примере имеется следующая цепочка вызовов функций:

main( ) → screen_refresh( ) → myshape :: draw( ) → rectangle :: draw ( ) →
put_line(a, b) → put_point(x, y) → on_screen(x, y).

Выход точки за пределы буферного массива *SCREEN* (экрана) выявляется функцией *on_screen*( ). Блок контроля вокруг вызова этой функции (или

вызывающей её *put_point*) не имеет смысла: на этом уровне ничего, кроме выдачи сообщения об ошибке, сделать нельзя, а такое сообщение можно выдать и непосредственно, не прибегая к механизму *throw — cath*. На уровне *main*( ) или *screen_refresh*( ) обрабатывать ошибку поздно, здесь можно только прервать выполнение программы; без серьёзной доработки класса *shape* содержательное сообщение об ошибке получить нельзя. В то же время блок контроля внутри функции *myshape* :: *draw*( ) позволит локализовать ошибку при выводе прямоугольника — контура фигуры *myshape* и, возможно, попробовать изменить его размер. В общем случае проектирование реакции программной системы на ошибки должно выполняться одновременно с проектированием её самой. Так, в программе, рассмотренной в учебном примере, можно снабдить каждую фигуру автоматически формируемым порядковым номером, значение которого можно выводить как часть сообщения об ошибке «выход за пределы экрана».

Если ошибка выявлена в конструкторе фигуры, фигура не создаётся. Это не является проблемой для цепочки базовых классов-значений. Исключение — класс shape, управляющий цепочкой фигур для рисования. Этот класс должен иметь деструктор, удаляющий сбойную фигуру из цепочки или заменяющий её специальной фигурой — значком ошибки. Деструктор для класса *shape* должен быть виртуальным, чтобы правильно удалять объекты всех производных классов.

Классы ошибок могут образовывать иерархию. В этом случае можно перехватом ошибки базового класса перехватить и все производные от него. Рекомендуется использовать в качестве базы исключений стандартное исключение *exception* (потребуется директива *#include <exception>*). Это позволит подключить программу к системному механизму перехвата исключений. Кроме того, можно использовать имеющийся в классе механизм для сообщений. Конструктор класса exception имеет аргумент типа *char** — строку (в стиле Си) для сообщения об ошибке. В блоке *catch* эта строка может быть получена вызовом виртуальной функции-члена *what*( ).

Например, класс ошибки перемещения фигуры может быть объявлен так:

```
struct CantBeMoved : std::exception
{
  CantBeMoved(const std::string& s) : std::exception(s.c_str( )) { }
};
```

Возбуждение исключения:

```
throw CantBeMoved("Line can't be moved: out of screen");
```

Перехват:

```
catch (CantBeMoved &ex) {  std::cout << ex.what() << "\n\n";  }
```

2.1. Практикум по теме

Переработать программу работы с библиотекой фигур, дополнив её механизмом контроля исключительных ситуаций. Возможно выявление следующих ошибок:

— непопадание точки на экран;

— некорректные параметры при формировании фигуры;

— нехватка места на экране для размещения фигуры в одной из позиций (исходной, повёрнутой, отражённой, перемещённой);

— повторный поворот/отражение уже повёрнутой/отражённой фигуры и т. п.

Организовать перехват исключений следует таким образом, чтобы искажения итоговой картинки были минимальны. Протестировать исключительные ситуации, результаты эксперимента поместить в отчёт.

Тема 3. КОМБИНИРОВАННЫЕ СТРУКТУРЫ ДАННЫХ И СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

В практических задачах множество часто используется как словарь. Основные операции со словарём — это поиск, добавление и удаление элементов множества (ключей). Двуместные операции со словарями выполняются сравнительно редко. При хранении множеств в форме упорядоченной последовательности операции поиска/вставки/удаления выполняются за линейное время. Если нужен только поиск, данные можно хранить в упорядоченном векторе (длиной n) с доступом к ключам за время $O(\log n)$. Для хранения словарей с возможностью пополнения применяются структуры данных, совмещающие быстрый поиск с быстрым добавлением и удалением ключей. Для небольших универсумов задачу решает вектор битов. Если же универсум велик, используются хеш-таблицы, деревья двоичного поиска и им подобные структуры данных.

3.1. Хеш-таблицы

Хеш-таблица — это обобщение способа хранения множества целых чисел (ключей) в форме вектора битов на случай, когда мощность универсума U велика по отношению к мощности множеств, с которыми нужно работать. Функция отображения преобразует значения ключей к интервалу $[0, m - 1]$, где m — размер хеш-таблицы, $m \ll |U|$. Очевидно, что при этом каждому индексу хеш-таблицы будет соответствовать много различных значений ключей. Поэтому, во-первых, в хеш-таблице приходится хранить не биты, а сами значения ключей, а во-вторых, имеется возможность размещать в ней более одного ключа для каждого значения функции отображения (разрешать коллизии).

Количество возможных коллизий можно уменьшить, если выполнить два условия:

1) выбрать размер хеш-таблицы с запасом. Если размер таблицы превышает мощность хранимого множества более чем вдвое, вероятность коллизии становится меньше 0,5. Если мощность множества заранее неизвестна, то выбирают некоторый начальный размер, а когда его оказывается недостаточно, таблицу перестраивают с увеличением размера (обычно вдвое);

2) подобрать функцию отображения (хеш-функцию) такую, чтобы все ячейки таблицы были востребованы по возможности с равной вероятностью,

независимо от того какое распределение имеют хранящиеся в таблице ключи.

По способу разрешения коллизий различают хеш-таблицы двух типов:

1) с открытой адресацией. Конфликтующие значения ключей размещаются в свободных ячейках таблицы;

2) с цепочками переполнения. Каждая ячейка таблицы содержит указатель на список конфликтующих ключей.

Подробнее о хеш-таблицах *см*. [1, с. 115–128], [13, с. 529–556], [5, с. 316–338].

Таблицы второго типа применяются чаще, потому что для них не существует проблемы переполнения. Если мощность хранимого множества становится слишком большой, таблица просто начинает работать как m списков. Если же таблица правильно построена и не переполнена, проверка принадлежности элемента множеству, а также вставка и удаление элемента выполняются в ней за постоянное время, примерно такое же, как и в массиве

103	102	101	100	35	50	–	32	31	30	–	44	123	90	105	120
55	38	37	20	–	–	–	80	–	–	–	60	–	10	–	104
–	70	–	–	–	–	–	–	–	–	–	–	–	–	–	40
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Рис. 3.1. Хеш-таблица из 16 экстентов с цепочками переполнения

битов (рис. 3.1).

За постоянное время (порядка размера таблицы) будут выполняться и двуместные операции над множествами: объединение, пересечение и разность: если хеш-функции для обоих множеств одинаковы, для этого пригоден такой же алгоритм попарного сравнения соответствующих ячеек, как и для массивов битов. В общем случае двуместная операция, организованная как просмотр первой таблицы, поиск каждого ключа во второй и вставка при необходимости в третью, имеет линейную сложность.

В хеш-таблице можно хранить и множество с повторениями: совпадающие значения ключей не создают никаких проблем, кроме гарантированных коллизий, которые разрешаются обычным образом.

К сожалению, всегда можно подобрать такие данные, что они все попадут в одну или несколько ячеек таблицы, образовав неупорядоченные списки (упорядочивание обычно не применяется). Это худший случай, для которого справедливы оценки временной сложности для списков: $O(n)$ — для поиска и удаления элемента; $O(n^2)$ — для двуместной операции над множествами.

Подбор подходящей хеш-функции — в общем случае достаточно сложная задача. Но если ключи представляют собой целые числа (или сводятся к таковым), хорошие результаты можно получить с хеш-функцией вида

$$h(x) = (a * x + b) \% m,$$

где m – размер таблицы, а a и b — простые числа.

Обычно a выбирается близким по значению к m, а b — к 1. Так, при $m = 100$ можно взять $a = 97$, $b = 11$. Такой выбор обеспечивает равномерное использование всех ячеек таблицы в большинстве практических случаев. Если размер таблицы m предполагается изменять, можно взять в качестве a любое достаточно большое простое число.

3.5. Поддержка произвольной последовательности в структуре данных для множеств

К одной структуре данных могут применяться операции как для множества, так и для последовательности. Иногда требуется поддерживать в структуре данных для множеств произвольную последовательность элементов этих множеств. Например, можно фиксировать порядок появления элементов в множестве при его создании и работать с этим порядком.

Последовательность из структуры данных для множеств может быть получена как результат её обхода. Часто этого бывает достаточно: порядок элементов для результата операции над множествами можно назначить произвольно. Однако для множества мощностью n это будет только одна из $n!$ возможных последовательностей. Если нужно поддерживать любую последовательность, возможны следующие подходы:

1) присоединить к каждому ключу дополнительное поле для хранения порядкового номера. Способ не создаёт проблем при поиске номера по значению ключа и при вставке новых ключей, они просто нумеруются по порядку. Удаление ключа требует просмотра всей структуры данных для корректировки номеров, следующих за удаляемым. То же приходится делать при поиске ключа по порядковому номеру (сложность $O(n)$);

2) с помощью дополнительных указателей для каждого ключа сформировать из них список, возможно, двунаправленный. Проходом по этому списку можно как восстановить хранящуюся последовательность, так и получить номер для каждого ключа. Доступ к ключу по номеру и наоборот в этом случае имеет линейную сложность. Зато как вставка, так и удаление ключа требуют минимальных накладных расходов;

3) создать массив указателей на ключи. Если одновременно поддерживать в ключах дополнительное поле с обратным указателем на соответствующие элементы массива, можно избежать дополнительных расходов как для определения ключа по номеру, так и номера по ключу. Недостаток способа —

необходимо заранее знать объём памяти для создания массива и перемещать часть массива в случае удаления ключей.

Операции над последовательностями, в отличие от операций с множествами, могут приводить к появлению дубликатов ключей. Структура данных для множеств должна обеспечивать соответствующую возможность.

Операции над последовательностями:

1. Слияние (*MERGE*). Объединение двух упорядоченных последовательностей в третью с сохранением упорядоченности. От операции объединения множеств отличается только возможностью появления дубликатов ключей. Если исходные последовательности не упорядочены, можно после их слияния просто упорядочить результат. Исходный порядок ключей в последовательностях в результате не сохраняется.

2. Сцепление (*CONCAT*). Вторая последовательность подсоединяется к концу первой, образуя её продолжение.

3. Размножение (*MUL*). Последовательность сцепляется сама с собой заданное количество раз.

4. Укорачивание (*ERASE*). Из последовательности исключается часть, ограниченная порядковыми номерами от $p1$ до $p2$.

5. Исключение (*EXCL*). Вторая последовательность исключается из первой, если она является её частью.

6. Включение (*SUBST*). Вторая последовательность включается в первую с указанной позиции p. Операция похожа на конкатенацию. Сперва берётся начало первой последовательности до позиции p, затем идёт вторая последовательность, а за ней — остаток первой.

7. Замена (*CHANGE*). Вторая последовательность заменяет элементы первой, начиная с заданной позиции p.

Пример. Пусть имеются две последовательности A = <5, 3, 2, 4, 6, 7, 9, 1> и B = <6, 7, 9>. Позиции считаются от 0.

Тогда операция *A.MERGE*(*B*) даст результат <1, 2, 3, 4, 5, 6, 6, 7, 7, 9, 9>;

A.CONCAT(*B*) — <5, 3, 2, 4, 6, 7, 9, 1, 6, 7, 9>;

B.MUL(3) — <6, 7, 9, 6, 7, 9, 6, 7, 9>;

A.ERASE(2, 4) — <5, 3, 7, 9, 1>;

A.EXCL(*B*) — <5, 3, 2, 4, 1>;

A.SUBST(*B*, 3) — <5, 3, 2, 6, 7, 9, 4, 6, 7, 9, 1>;

A.CHANGE(*B*, 2) — <5, 3, 6, 7, 9, 7, 9, 1>.

3.7. Использование стандартной библиотеки шаблонов

Стандартная библиотека шаблонов (*STL*) поддерживает большинство типовых операций со структурами данных.

Мы уже использовали последовательные контейнеры *vector*, *list* и *deque* и их производные (адаптеры) *stack* и *queue*. В STL имеются ассоциативные контейнеры: *set* для множеств, *map* для отображений, *multiset*, *multimap* — для множеств и отображений с повторениями, основанные на деревьях двоичного поиска, и их аналоги *unordered_set*, *unordered_map*, *unordered_multiset* и *unordered_multimap* на базе хеш-таблиц. Для обработки данных используются возможности библиотеки алгоритмов (*algorithm*).

Каждому контейнеру соответствует заголовочный файл, который нужно подключать директивой #*include*.

Контейнеры *set* и *map* хранят множества в виде дерева двоичного поиска с автобалансировкой (красно-чёрное дерево). Контейнер *set* хранит множество ключей, а *map* — пары <ключ, значение>, причём все ключи в них уникальны. Для множеств с повторениями используются контейнеры *multimap* и *multiset*. При просмотре всех этих контейнеров их содержимое выдаётся в виде упорядоченной последовательности (внутренний обход дерева двоичного поиска).

При просмотре *unordered* контейнеров будет выдана неупорядоченная последовательность ключей.

Возможно много вариантов приспособления контейнеров для работы с последовательностями: использование *map* (или *multimap*) вместо *set*, чтобы хранить вместе с ключами их порядковые номера, комбинирование контейнера для множеств с контейнером последовательностей (*vector* или *forward_list*), хранящим итераторы, и т. п.

Конструкторы контейнеров позволяют уже при их объявлении сформировать множество заданной мощности. Для этого достаточно в качестве инициализатора содержимого контейнера использовать датчик случайных чисел.

Всё необходимое для операций с контейнерами можно найти в библиотеке алгоритмов (*algorithm*). В частности, в ней имеются функции *set_union*, *set_intersection*, *set_difference*, *set_symmetric_difference*, вычисляющие объединение, пересечение, вычитание и симметрическую разность множеств. Функции принимают в качестве аргументов отрезки из двух контейнеров и формируют новый контейнер с результатом. В них реализуется схема слияния, поэтому входные отрезки должны быть

упорядочены. Это справедливо по умолчанию для контейнеров *set, map* и аналогичных. Для *unordered_set* аналогичные результаты даёт одновременный просмотр двух контейнеров с применением функций проверки наличия и вставки элемента множества в результат. В библиотеке *STL* имеются функции для выполнения любых операций с последовательностями.

Подробнее *см*. [3, с. 295–368], [11, с. 835–962]. Полезно также посмотреть библиотечные файлы в каталоге *include* компилятора C++: только там содержится исчерпывающая информация о том, какие на самом деле объявляются классы и какие функции-члены они содержат. Информация в литературных источниках, как правило, запаздывает и содержит неточности. Важно и то, что в сообщениях компилятора об ошибках обычно присутствует информация из текстов каталога *include*, поскольку эти тексты компилируются вместе с программой пользователя.

3.8. Превращение в контейнер пользовательской структуры данных

Пользовательскую структуру данных для хранения множества/последовательности можно превратить в подобие библиотечного контейнера и тем самым обеспечить возможность применения к нему стандартных алгоритмов библиотеки *algorithm*. Для этого достаточно соблюдать соглашения о кодировании, принятые для контейнеров: объявить классы итераторов для просмотра и вставки и функции для их инициализации и контроля.

Можно ограничиться только теми средствами, которые действительно понадобятся для работы с пользовательским контейнером.

Для просмотра контейнера в цикле необходимы и достаточны функции *begin*( ) и *end*( ), возвращающие прямой итератор чтения, указывающий на первый элемент контейнера и элемент «сразу за последним» соответственно. Итератор должен поддерживать операцию разыменования для доступа к элементам контейнера, операцию инкремента для перемещения по контейнеру и операцию сравнения с результатом end( ). Для вставки нового значения необходим итератор вставки и средство для его инициализации (инсертер).

Наличие этих средств позволит, например, стандартным образом получить копию не только стандартного, но и пользовательского контейнера *A* в произвольном контейнере *B*:

```
std::copy(A.begin( ), A.end( ), back_inserter(B));
```

Пример объявления итераторов для пользовательского контейнера (хеш-таблица из массива *bct*, содержащего указатели на цепочки переполнения).

```
#include <iterator>
using namespace std;
//ИТЕРАТОР ЧТЕНИЯ — нужны сравнения, разыменования, инкремент
struct myiter : public std::iterator<std::forward_iterator_tag, int>
{ //В качестве базы использован стандартный прямой итератор
  myiter(Node *p) : bct(nullptr), pos(0), Ptr(p) {}
  bool operator == (const myiter & Other) const { return Ptr ==
Other.Ptr; }
  bool operator != (const myiter & Other) const { return Ptr != Other.Ptr;
}
  myiter operator++(); //Ключевая операция — инкремент по контейнеру
  myiter operator++(int) { myiter temp(*this); ++*this; return temp; }
  pointer operator->() { return & Ptr->key; } //Разыменование косвенное
  reference operator*() { return Ptr->key; } //Разыменование прямое
  //protected:
// Container& c;
  Node **bct; //Указатель на хеш-таблицу (массив экстентов)
  size_t pos; //Номер текущего экстента
  Node * Ptr; //Реальный указатель на элемент контейнера
};
//ИТЕРАТОР ВСТАВКИ — нужно только присваивание!
template <typename Container, typename Iter = myiter>
class outiter : public std::iterator<std::output_iterator_tag, typename
Container::value_type>
{
protected:
  Container& container;    // Контейнер для вставки элементов
  Iter iter;      // Текущее значение итератора чтения
public:
  // Конструктор
  explicit outiter(Container& c, Iter it) : container(c), iter(it) { }
  // Присваивание = вставка ключа в контейнер
  const outiter<Container>&
        operator = (const typename Container::value_type& value) {
        iter = container.insert(value, iter).first;
        return *this;
  }
```

```
    const outiter<Container>& //Присваивание копии — фиктивное
        operator = (const outiter<Container>&) { return *this; }
    // Разыменование — пустая операция
    outiter<Container>& operator* ( ) { return *this; }
    // Инкремент — пустая операция
    outiter<Container>& operator++ ( ) { return *this; } //префиксный
    outiter<Container>& operator++ (int) { return *this; } //постфиксный
};
// ИНСЕРТЕР — функция для создания итератора вставки — аргумент для алгоритма, создающего новый контейнер (универсальная)
template <typename Container, typename Iter>
inline outiter<Container, Iter> outinserter(Container& c, Iter it)
{  return outiter<Container, Iter>(c, it); }

//Функции, превращающие хеш-таблицу в контейнер
myiter HT::begin( )const { //Получение итератора на начало
    myiter p(nullptr); //Инициализация итератора значением «на конец»
    p.bct = bucket; //Установка на массив экстентов хеш-таблицы
    for (; p.pos < Buckets; ++p.pos) { //Поиск первого элемента в XT
        p.Ptr = bucket[p.pos];
        if (p.Ptr) break; //Нашли!
    }
    return p;
}

myiter HT::end( )const { return myiter(nullptr); } //Итератор на конец

myiter myiter::operator++( ) //Инкремент итератора: пример для XT
{
    if (!Ptr) { //Инициализация сделана?
        return *this; //Не работает без предварительной установки на XT
    }
    else
    { //Текущий уже выдан
        if(Ptr->down) { //Есть следующий, вниз
            Ptr = Ptr->down;
            return (*this);
```

```
            }
               while(++pos < HT::Buckets)       //Поиск очередного элемента
                     if(Ptr = bct[pos]) return *this; //Нашли — выход
               }
            Ptr = nullptr; //Не нашли — выход с пустым итератором
            return *this;
      }
   }
```

Детальную информацию о пользовательских итераторах уровня стандарта C++17 *см*. в источниках [2], [3] и [10]. Справку о текущем состоянии вопроса можно получить в интернете на сайте ru.cppreference.com.

3.9. Практикум по теме

Реализовать индивидуальное задание темы «Множества+последовательности» в виде программы, используя свой контейнер для заданной структуры данных (хеш-таблицы или одного из вариантов ДДП), и доработать его для поддержки операций с последовательностями. Для операций с контейнером рекомендуется использовать возможности библиотеки алгоритмов. Программа должна реализовывать цепочку операций над множествами, имеющимися в выражении, взятом по номеру варианта задания из табл. 3.1 с базовым контейнером и операциями с последовательностью из табл. 3.2. Результат каждого шага цепочки операций выводится на экран.

Таблица 3.1

Индивидуальные задания к теме «Комбинированные структуры данных». Операции над множествами

№ варианта	Мощность множества	Что надо вычислить
8	26	A \ (B ∩ C) ∪ D ⊕ E

Таблица 3.2

Индивидуальные задания к теме «Комбинированные структуры данных». Базовая структура данных и операции над последовательностями

№ вари-анта	Базо-вая СД	Дополнительные операции
8	XT	MERGE, ERASE, EXCL

Примечание. В таблице использованы следующие обозначения:

XT — хеш-таблица

Итератор чтения для просмотра структуры данных или её части можно реализовать для всех перечисленных выше структур данных. Итератор вставки, обеспечивающий добавление ключей в множество, может без проблем быть реализован только для хеш-таблицы. Указание места вставки для неё игнорируется как совершенно излишнее. Для ДДП вставка без указания места начала поиска может быть выполнена только за логарифмическое время. Вставка за константное время возможна, только если итератор вставки укажет для начала поиска места вставки на ключ, вставленный последним. В случае вставки произвольного ключа такое указание приведёт к хаосу. Оно допустимо и целесообразно только для вставки упорядоченной последовательности ключей в пустое дерево, как это имеет место в двуместных операциях с множествами по схеме слияния. Чтобы форма ДДП такими вставками не искажалась, оно должно быть с автобалансировкой. Чтобы обеспечить двуместную операцию за линейное время с обычным ДДП — без автобалансировки, необходимо помещать результат операции в промежуточный буфер (вектор) и затем восстанавливать дерево из него.

ДДП с хранением в узлах высоты работает аналогично самобалансирующемуся АВЛ-дереву: балансы вычисляются динамически сравнением высот поддеревьев.

Для деревьев с автобалансировкой итератор вставки обеспечивает в качестве точки начала очередного поиска узел, на котором балансировка закончилась. Для этого в нём хранится стек с путём от корня к точке вставки.

В ДДП с хранением в узлах мощности поддеревьев само по себе не имеет преимуществ перед обычным ДДП при случайных вставках. Но при двуместных операциях итератор вставки может выполнять вставку нового узла в корень (за константное время), а по завершении операции результат балансируется, как и для обычного ДДП, но без использования

дополнительной памяти.

Для 2-3-дерева возможна как схема с автобалансировкой после каждой вставки, так и получение промежуточного списка упорядоченных узлов, из которого по завершении двуместной операции дерево восстанавливается за линейное время без использования дополнительной памяти. Узлы дерева при балансировке не перемещаются, и итераторы на них остаются действительны.